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:

```
#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.

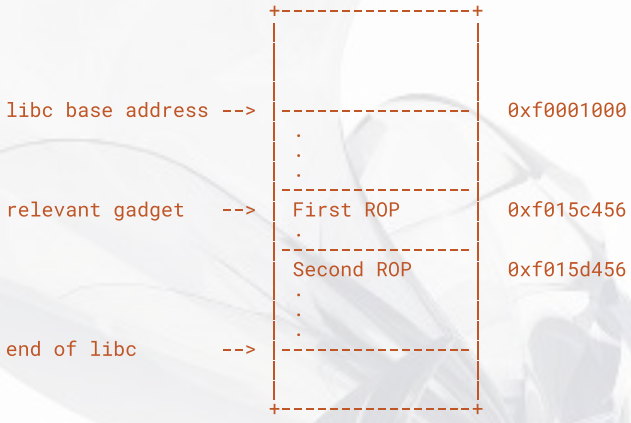
And here is how the relevant chunk of memory in libc looks:

```
                      +-----------------+
                      |                 |
                      |                 |
                      |                 |
                      |                 |
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!

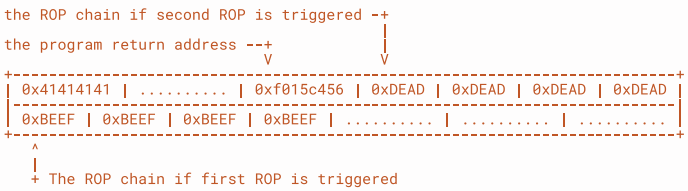
In order to successfully execute arbitrary code in both cases, we need to know which ROP gadget was triggered - which tells us what the libc base address is. Once we figure that out, we can continue crafting the rest of our chain accordingly.

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!

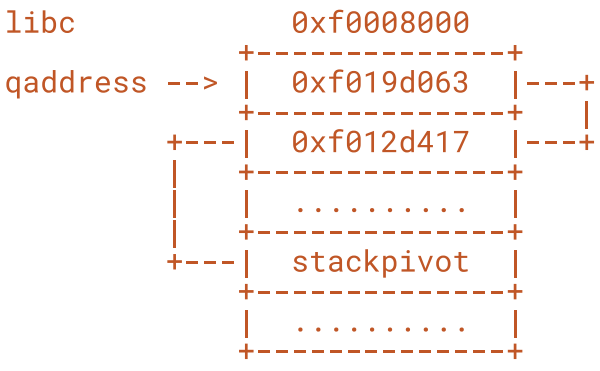
This is a huge deal. ASLR is not as effective as we used to think.

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/