Blind Return Oriented Programming

tl;dr

In 2014 a paper [http://www.scs.stanford.edu/brop/bittau-brop.pdf] which introduces Blind Return Oriented Programming (BROP), a state-of-the-art exploitation technique, was released by researchers from Stanford University. The paper discusses a general approach in which BROP is used to exploit services which are both vulnerable to stack-based buffer overflows and automatically recover after a crash. What is best about the BROP technique is that one does not need to possess the binary and the source code of the target service to be able to successfully exploit it.

I have tried to replicate the research and exploit the Nginx and MySQL vulnerabilities using BROP to better understand the technique; unfortunately, I have found that the tool released by the researchers failed to exploit either bug in my test environment. The idea is brilliant and works in theory, and the code was clearly working for them in their test environments. What could be the difference(s) that resulted in the same vulnerabilities not being exploitable by the tool? In this blog post we will have a look at some important steps of the exploitation technique.

For those not familiar with the BROP technique, it is highly advised that you read the previously linked BROP paper first, as this post assumes the reader is familiar with the idea and the process.

Stack Reading

Stack reading is based on the idea that one can read values from the stack by brute-forcing each byte position of the value. If a correct value is found for the given byte position the application will continue to run, otherwise it will crash. This allows reading the stack canary (if it is there), data, code pointers, and so on. This technique heavily relies on the aforementioned requirement that the target service will gracefully recover from crashes – typically this simply involves the service forking a new process to handle each incoming request.

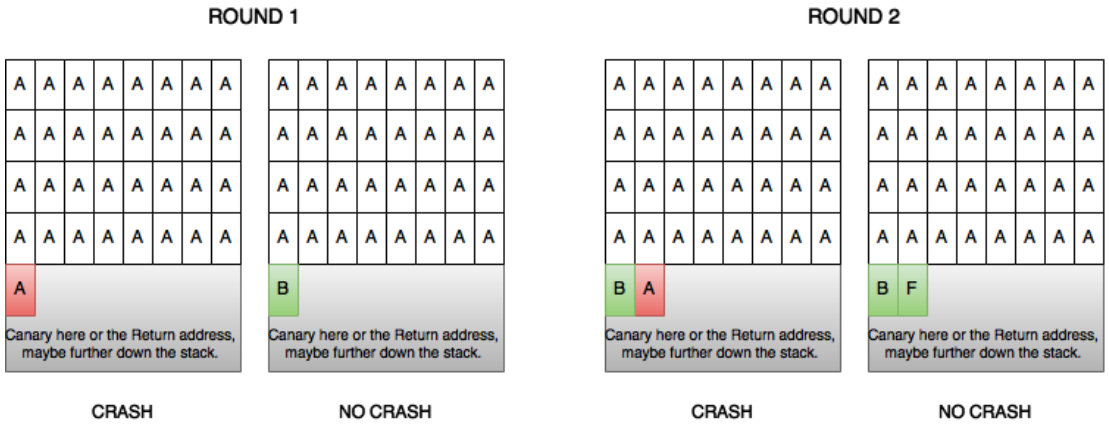


Figure 1: Stack reading

As illustrated in the diagram above, by stack reading we can eventually brute-force the stack canaries, therefore bypassing the mitigation if the canary does not change between crashes; furthermore, we can read the return address which results in an ASLR bypass. However, while developing the exploit for MySQL and Nginx I had to face several difficulties, explained below.

Unusual Buffer Handling

The first difficulty addressed by the proof-of-concept code provided by the researchers was that the MySQL vulnerability involved an overrun buffer that was populated backwards – in the opposite direction from that to which you would typically expect a stack buffer to be written. Because of this quirk, the payload had to be reversed before sending it to MySQL. This backwards copy can best be explained using the diagram below.

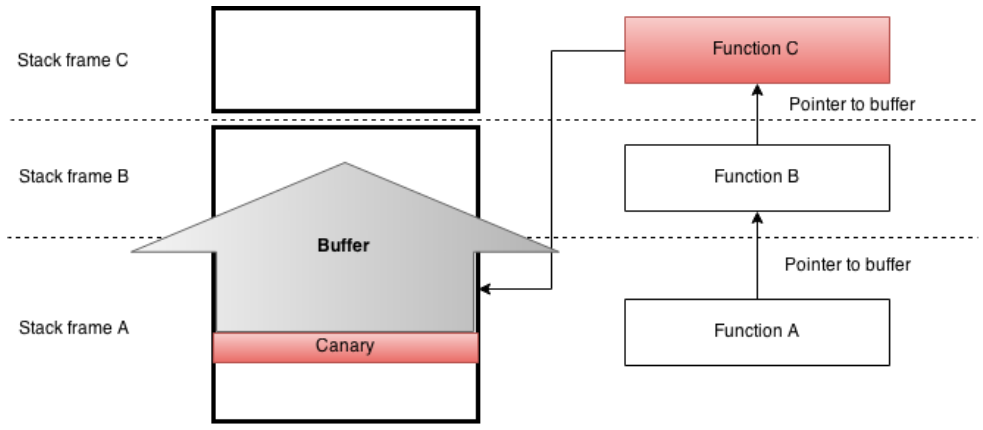


Figure 2: MySQL Buffer Overflow

A quick investigation revealed that function B received a pointer to a buffer allocated by its caller. Function B passed this pointer to function C, which started reading data into the buffer starting from the end of the buffer. Function C did not check the size of the buffer and started to overwrite data in stack frame B; this included the return address of function B. In terms of exploitation this makes things quite easy, as neither function B nor function C allocated anything on the stack, therefore the stack frame of these functions did not contain a stack canary. Even if there had been a stack canary, it would have been possible to read it. To summarise: the return address to be overwritten was not after, but before, the buffer. Despite this, it was still possible to read the stack of both Nginx and MySQL with some small modifications.

Return Address Location

The second issue I had to face was more challenging, and probably this is the point where I will contribute something useful to future use of the BROP technique. The BROP paper mentioned that one has to read three values (including the canary) to get the return address. It might have been the case for the authors of the paper, as I've illustrated on the left side of the diagram below, but my version of the services, compiled on Ubuntu, proved otherwise.

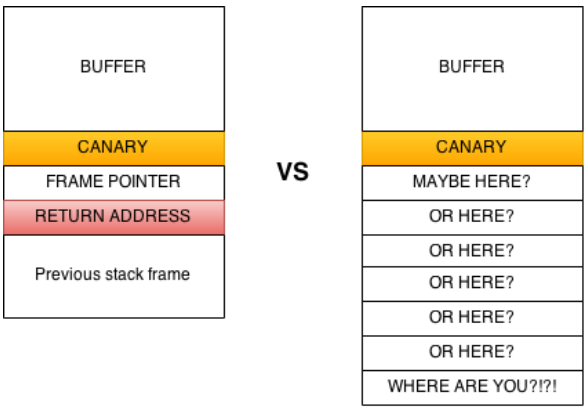


Figure 3: Return Address Location

As shown above, in my case (on the right) the return address was not at the location where the proof-of-concept code expected it. As a result, the code failed at the very early stages of the exploitation.

The first diagram illustrated in the official StackGuard stack protection paper [ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf] shows that there are saved register values between the canary and the return address. I've included a diagram similar to the one in their paper below.

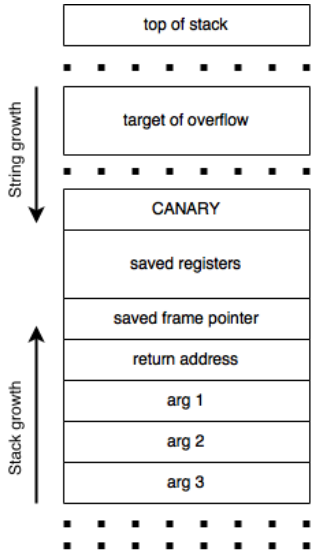


Figure 4: StackGuard Stack Layout

What's illustrated above pretty much matched my scenario, and thus raises the question: how do we find out where the return address is? The original researchers' BROP proof-of-concept used patterns to detect whether ASLR is enabled or not, and to determine the nature of some of the values on the stack. I thought I might be able to reuse this to find the return address. The code assumed the following, described with pseudocode:

```
    if address > 0x400000 and address < 0x500000: CODE – NO ASLR
    if (address & 0x7fff00000000) == 0x7fff00000000: STACK FRAME POINTER
    if (address & 0x7f0000000000) == 0x7f0000000000: CODE - ASLR
```

While experimenting with this code I encountered, and thought of, a number of problems with the assumptions made. Some of these manifested in the form of both false positives and false negatives. First, depending on the size of the target service it might happen that in case ASLR protection is disabled, the return address is actually greater than 0x500000, which was the case with the MySQL binary I was exploiting. Second, what if for example the value 0x4e4033 is just a simple numeric value pushed onto the stack for some reason? Finally, in my testing the third pattern matched both data and code pointers when ASLR was enabled.

I had to rework the patterns and implement a method to find the return address with much higher accuracy. This resulted in a multi-step process.

First, I had to make some changes to the patterns, which can be seen below.

```
if address < 0x400000: NOT RELEVANT
if address >= 0x7fff00000000: NOT RELEVANT
if address > 0x400000: MAYBE
if (address & 0x7f0000000000) == 0x7f0000000000: MAYBE
```

According to this, if we would like to eliminate values not relevant for us we can use the following simple pattern. If an address matches the pattern it is a return address **candidate**.

```
if address > 0x400000 and address < 0x7fff00000000: RETADDR CANDIDATE
```

The first step of the actual process was to read several values from the stack (in my testing reading ten values was safe) and check each of them against the pattern above. The values read from the stack when testing Nginx, along with their position on the stack (starting from the very beginning of the buffer) can be seen below.

```
ADDRESS                 STACK POS
0x3539952936a15a00      513
0xffffffffffffffffL     514
0x7f28fa9df750          515
0x7f28fa9ea0f4          516
0xfefefefefefefefeL     517
0xfefefefefefefefeL     518
0x7f28f9e894fe          519
0xffffffffffffffffL     520
0x7f28f9ebfbff          521
0x7f28f9ecffff          522
```

The canary has some properties which makes it easy to recognize above, especially if we take into consideration the context. One of these properties obviously is its location on the stack. Another one is that it seems to be random and does not look like a pointer. Also, the last byte of the canary is always 0x00. For more information about stack canaries please refer to the StackGuard paper mentioned earlier.

Right after there is a non-relevant value followed by one which seems to look like a return address. Trust me, it was not.

After applying the first step on the dumped values I have received the following reduced list of return address candidates.

```
STACK AFTER 1st STAGE CLEANUP (pattern)

    ADDRESS           : STACK POS       CRASH
    0x7f28fa9df750    : 515             0
    0x7f28fa9ea0f4    : 516             0
    0x7f28f9e894fe    : 519             0
    0x7f28f9ebfbff    : 521             0
    0x7f28f9ecffff    : 522             0
```

There were still far too many return address candidates, though at least one of them is a return address. I have implemented a second stage check which actively measures the sensitivity of each value. This is done by adding and subtracting values from the address and incrementing a crash counter variable by one for each crash. This is because stack positions which do not cause a crash whatever the value is will have a crash counter value of zero. Data pointers still can cause crashes but are generally less sensitive. Please note that it might be that data pointer tampering results in more crashes. Such cases include when the return address is tampered with and points to a valid instruction (not misaligned), and skipping one or more instructions will not result in a crash. According to what was written above, once the crash count is set for each stack item, the list of items should be ordered by the crash count in decrementing order. Items with a crash count of zero can safely be removed from the list. This leaves the following smaller list:

```
STACK AFTER 2nd STAGE CLEANUP (crash_count)

    ADDRESS           : STACK POS       CRASH
    0x7f28f9e894fe    : 519             6
    0x7f28f9ebfbff    : 521             3
```

A third stage was also implemented for cases when the list after the second stage still contains both data and code pointers.

The third stage builds on the idea that the data segment is usually at higher memory addresses than the code. One reason for this is that if the code is buggy and there is a buffer overflow it will not be possible to overwrite code. The first step of the process is to get the smallest and biggest addresses from our list, then calculate the distance between them. If we have more than three values remaining on our list and the greatest distance is above 0x200000 (the 0x200000 distance is not fixed either and probably should not be considered as a general rule, but it worked in my testing), indicating that both data and code pointers are present, only values which are closer to the smallest address should be kept in the list. This is because we assume (and this is the case most of the time as mentioned earlier) that data is located at higher memory addresses than code. This stage will leave us with a list free of data pointers.

The fourth stage is simple list ordering. First, we order the list based on the crash count, then, based on their position on the stack. As mentioned earlier, the most sensitive address is more likely to be a return address. At this stage we might still have multiple return address candidates even with the same crash count value. The value closer to the beginning of the buffer is a better choice, as chances are that the first return address is the one belonging to our stack frame.

In my case the list resulting from the fourth stage was identical to the second stage shown earlier. And, indeed, the value 0x7f28f9e894fe at stack position 519 was the return address.

This process is not bulletproof, but still proved to be reliable in my testing of multiple builds (PIE and Non-PIE builds with different optimisation levels) of both MySQL and Nginx.

Finding a STOP Gadget

Finding a stop gadget is a straightforward process. However, using an infinite loop type of stop gadget for everything, especially in the case of Nginx, seems to be a bad idea, as we will kill all the workers quickly.

As the stop gadget is only used as a signalling mechanism, just as the original paper says, it can be anything which results in a detectable signal. For example, writing on a socket, sleeping for an amount of time, or making the service continue to serve requests.

The best STOP gadget searching idea in the paper was to get the service to "jump" back into the main worker loop. To do this we have to locate a gadget that simply returns (return gadget), or find the PLT, take a PLT entry that returns, and start padding the stack with the address of the return gadget or the PLT entry one stack position per iteration while inspecting service behaviour. Once we pad the stack with the address of the PLT entry to reach the return address to the main worker loop the service should resume serving requests.

There are two problems with this. First, if we are looking for a return gadget, how do we know whether the address being probed actually points to a return gadget or not? The service will crash in both cases. Second, the process of finding the PLT might not require the use of a stop gadget if:

- we are lucky enough not to hit an infinite loop type of stop gadget before reaching the PLT
- if we hit a PLT entry suitable to be used as a stop gadget (more about this later) we have to be lucky enough to have register value(s) which will make the PLT entry produce a detectable behaviour.

Otherwise, the PLT detection pattern requires the use of a stop gadget: without a stop gadget we will not be able to craft working detection patterns.

Even if we manage to find a return gadget, chances are high that the idea of returning to the main worker loop will not work. For me, in the worst cases the service ended up in an infinite loop; most of the time it just crashed but unfortunately never resumed serving requests. Also, the MySQL buffer did not allow for padding the stack to the main loop entry. This is because of how the buffer is handled (Figure 2).

PLT entries such as `usleep`, `nanosleep`, write, and read could be used as stop gadgets as suggested earlier. The problem is that without a BROP gadget or at least a `POP RDI; RET` gadget this is, well, not really possible, except if we are lucky. Both in case of Nginx and MySQL I was not lucky enough to have a suitable pre-populated value in `RDI` (used to store the first argument to a function on x64) and so I could not trigger a detectable pause when I hit `usleep` or `nanosleep` in the PLT. In one case only `nanosleep` was available but the value in RDI was too small to be able to detect `nanosleep`, therefore it was not possible to use it as a stop gadget without finding a BROP gadget first. What this means is that at some point in the exploitation process, if we had to use an infinite type loop gadget, we can replace that with a safe version. This would be very useful in case of Nginx, but because of the use of the infinite loop type stop gadget we will probably kill all the workers before getting there. In the other case I had `usleep` in the PLT but the value of RDI was 0x0.

The problem was the same with the other PLT entries: the registers were not controllable at this point. This confirms what was written in the paper: better not to expect registers to have useful values.

I had tried several things to avoid using a stop gadget that triggers an infinite loop but none of them worked. This meant that I would always exhaust all worker threads of Nginx while trying to find the gadget. At this point I decided to shift focus primarily towards MySQL, hoping that if I made progress there the experience would help me come up with something that would help to make progress with Nginx.

In case of MySQL I have used an infinite-loop type stop gadget because the service was forking threads and there was no hard limit on the number of simultaneously running threads. Of course extensive use of this type of stop gadget can cause problems, for example eating up all sockets until no more connections are accepted. Because of this, I have implemented code to crash the whole process regularly, keeping the service accessible.

Finding a BROP Gadget

To find a BROP gadget, the original paper suggests the use of the following stack pattern.

```
probe, stop, stop, stop, stop, stop, stop, stop, traps
```

The problem with the pattern above is that it will find any gadget that pops up 0 to 6 values. Using the stack pattern above results in the following issues:

- The probe itself can point to an address where when the instructions get executed the service can go into an infinite loop. This is something I have encountered all the time while using the proof-of-concept code and with my own code (which used a better pattern, more on this later) as well.
- The above pattern will match all the gadgets in the list below, and any gadgets that pop the 6th value from the stack then return, or simply return to the address on the 7th stack position. From the list below only the last one is good candidate of being a BROP gadget. But all share one thing in common: once hit they cause an infinite loop. During the scanning process we can easily kill all four workers of Nginx before hitting a valid BROP gadget, unless we are lucky enough to find a stop gadget which does not trigger an infinite loop.
    - RET
    - POP <REG>; RET
    - POP <REG>; POP <REG>; RET
    - POP <REG>; POP <REG>; POP <REG>; RET
    - POP <REG>; POP <REG>; POP <REG>; POP <REG>; RET
    - POP <REG>; POP <REG>; POP <REG>; POP <REG>; POP <REG>; RET
    - POP <REG>; POP <REG>; POP <REG>; POP <REG>; POP <REG>; POP <REG>; RET

We cannot really avoid hitting an unwanted stop gadget with the probe, but we can avoid unnecessarily hitting the stop gadgets we have put on the stack as part of the pattern.

The pattern I was using to find a BROP gadget can be seen below.

```
probe, trap, trap, trap, trap, trap, trap, stop
```

The following list shows which gadgets the pattern will match:

- RET
- POP <REG>; POP <REG>; POP <REG>; POP <REG>; POP <REG>; POP <REG>; RET

The chances of hitting a stop gadget are greatly reduced using the new pattern. One thing to note though is that even if the probe itself will not point to a stop gadget and we hit the stop gadget we put on the stack, therefore a chain of instructions matching the pattern was found … yes, exactly. The pattern matches any chain of instructions which pops the 6th value from the stack then returns, or simply returns to the address on the 7th stack position.

To reduce false positives the paper mentions the following optimisation:

"A misaligned parse in the middle yields a pop rsp which will cause a crash and can be used to verify the gadget and further eliminate false positives"

The problem is, a misaligned parse with the same offset in the chain of instructions matching the BROP pattern based on behaviour can cause a crash too.

As an example I have set up a weak pattern as follows:

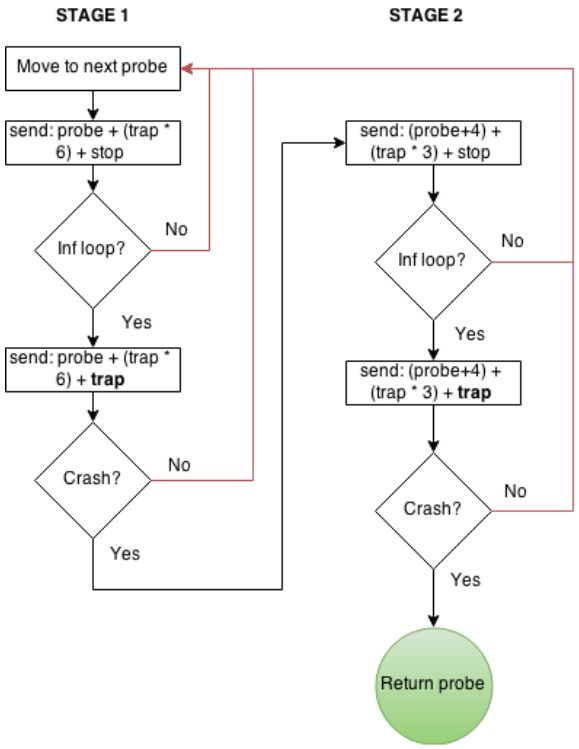
```
if is_brop_gadget([probe, trap, trap, trap, trap, trap, trap, stop]) == 2 and \
    not is_brop_gadget([probe, trap, trap, trap, trap, trap, trap, trap]) and \
    not is_brop_gadget([probe + 3, trap, trap, trap, trap, stop]):
                return probe
```

The code above is supposed to return the address where a BROP gadget was found. The arguments to the `is_brop_gadget` function are the patterns. `is_brop_gadget`, which returns 2 (TIMEOUT) in case of an infinite loop, False if the service crashed, and True if the payload did not cause a crash and the service is still running.

My code using the patterns above reported the address 0x4e7a52 to be pointing to a BROP gadget. If we have a look at the code at the address we will immediately see that it is not what we were looking for.

```
(gdb) x/16i 0x4e7a52
   0x4e7a52 <ftell@plt+2>:      xor    %dh,%dh
   0x4e7a54 <ftell@plt+4>:      jo     0x4e7a56 <ftell@plt+6>
   0x4e7a56 <ftell@plt+6>:      pushq  $0xe
   0x4e7a5b <ftell@plt+11>:     jmpq   0x4e7960
   0x4e7a60 <shutdown@plt>:     jmpq   *0x70f62a(%rip)        # 0xbf7090
<shutdown@got.plt>
   0x4e7a66 <shutdown@plt+6>:   pushq  $0xf
   0x4e7a6b <shutdown@plt+11>:  jmpq   0x4e7960
   0x4e7a70 <__strncpy_chk@plt>:        jmpq   *0x70f622(%rip)        # 0xbf7098
<__strncpy_chk@got.plt>
   0x4e7a76 <__strncpy_chk@plt+6>:      pushq  $0x10
   0x4e7a7b <__strncpy_chk@plt+11>:     jmpq   0x4e7960
   0x4e7a80 <close@plt>:        jmpq   *0x70f61a(%rip)        # 0xbf70a0
<close@got.plt>
   0x4e7a86 <close@plt+6>:      pushq  $0x11
   0x4e7a8b <close@plt+11>:     jmpq   0x4e7960
   0x4e7a90 <ceil@plt>: jmpq   *0x70f612(%rip)        # 0xbf70a8 <ceil@got.plt>
         0x4e7a96 <ceil@plt+6>:       pushq  $0x12
   0x4e7a9b <ceil@plt+11>:      jmpq   0x4e7960
```

The BROP gadget detection mechanism I have implemented can be seen below.



The process is made up of two stages. The first stage determines whether the address pointed to by our probe has the BROP gadget behaviour. This is done using the following two patterns:

- probe, trap, trap, trap, trap, trap, trap, **stop**
  This pattern is used to find code which behaves like a BROP gadget.
- probe, trap, trap, trap, trap, trap, trap, **trap**
  This pattern is used to detect whether the probe itself points to a stop gadget or the stop gadget hit is the one we have put on the stack. This is because if it was our stop gadget that hit when using the first pattern, replacing the stop gadget with a trap should result in a crash.

The problem obviously is if that it was the probe acting as a stop gadget, in case of Nginx, we have killed yet another worker.

Anyway, let's have a look at the second stage of the process, which is intended to reduce false positives in a more reliable way. The following patterns were used:

- probe + 4, trap, trap, trap, **stop**
  If a BROP gadget was hit and we add four to the probed address, only three values should be popped from the stack. We can confirm this by having a look at the BROP gadget below.

```
5b              pop    %rbx
5d              pop    %rbp
41 5c           pop    %r12
41 5d           pop    %r13    < PROBE + 4
41 5e           pop    %r14
41 5f           pop    %r15
                ret
```

- Probe + 4, trap, trap, trap, **trap**
  This pattern is used to detect whether it was our stop gadget hit or the service ended up in an infinite loop, which might happen if the new probe address (+4) triggers an infinite loop.

If all of these patterns are matched we can be pretty confident that a BROP gadget was found. Of course this method has several downsides, such as the increased request (therefore crash) count and the increase in time required to find the gadget, as scanning is done one byte a time, while it still can happen that unwanted stop gadgets get hit.

Just for the sake of completeness, below is the pattern my code was using to find a BROP gadget.

```
if is_brop_gadget([probe, stop]) != 2 and \
   is_brop_gadget([probe, trap]) == False and \
   is_brop_gadget([probe, trap, trap, trap, trap, trap, trap, stop]) == 2 and \
   is_brop_gadget([probe, trap, trap, trap, trap, trap, trap, trap]) == False and
\
   is_brop_gadget([probe + 4, trap, trap, trap, stop]) == 2 and \
   is_brop_gadget([probe + 4, trap, trap, trap, trap]) == False and \
   is_brop_gadget([probe + 3, trap, trap, trap, trap, stop]) == False:
          return probe
```

As can be seen, I added the check for the POP RSP crash as the last pattern. Of course the more patterns we use, the longer the process will take and the noisier the attack gets. The exploit using the patterns above reported the address as pointing to 0x4e8b19. Having a look at the address with a debugger, we can confirm that the address indeed points to a BROP gadget.

```
(gdb) x/7i 0x4e8b19
   0x4e8b19 <_ZL17fix_type_pointersPPPKcP10st_typelibjPPc.constprop.75+132>:
pop    %rbx
   0x4e8b1a <_ZL17fix_type_pointersPPPKcP10st_typelibjPPc.constprop.75+133>:
pop    %rbp
```

```
   0x4e8b1b <_ZL17fix_type_pointersPPPKcP10st_typelibjPPc.constprop.75+134>:
pop    %r12
   0x4e8b1d <_ZL17fix_type_pointersPPPKcP10st_typelibjPPc.constprop.75+136>:
pop    %r13
   0x4e8b1f <_ZL17fix_type_pointersPPPKcP10st_typelibjPPc.constprop.75+138>:
pop    %r14
   0x4e8b21 <_ZL17fix_type_pointersPPPKcP10st_typelibjPPc.constprop.75+140>:
pop    %r15
   0x4e8b23 <_ZL17fix_type_pointersPPPKcP10st_typelibjPPc.constprop.75+142>:
retq
```

Unfortunately, in case of Nginx all methods ended up rendering all the workers into an infinite loop. Even using the original pattern and skipping seven bytes per iteration while scanning resulted in hitting way too many unwanted stop gadgets. With MySQL, however, the method I have implemented was working fine.

Finding the Procedure Linkage Table

There can be multiple complications during the PLT lookup process. The very first issue I have found, again, relates to the detection patterns. According to the original paper one should use the probe, stop, trap pattern to check whether the stop gadget was hit and if it was hit, check probe+6, stop, trap. If both patterns match, one should check if the neighbouring entries behave the same way. If they do, the PLT was found. In my case there was plenty of code between the PLT and the ELF header, and unfortunately multiple addresses behaved as a stop gadget. The situation got even worse when the service was compiled with PIE, as the scanning is done towards lower memory addresses, meaning higher chances to hit stop gadgets.

I have used the following patterns to find the PLT:

```
        if is_PLT(probe) and \
           is_PLT(probe + 6) and \
           is_PLT(probe + 16) and \
           is_PLT(probe + 32) and \
           is_PLT(probe + 38):
            return True
```

In general, the more checks we perform, the better our chances are to find the PLT. I have found code at several locations which matched the patterns *probe, probe+6, probe+16 and probe+32* and only when *probe+38* was added did it turn out that I was not looking at a PLT entry. As each of these checks involved the use of a stop gadget, false positives may render all the workers of Nginx into an infinite loop. Fortunately the PLT lookup was working fine with MySQL.

Dealing with the PLT

Using the PLT can have unexpected consequences. A very obvious and fatal one will be mentioned here. Another, related to locating `write()` will be explained later.

It happened that while scanning the PLT for a specific PLT entry I hit `exit()`. While this would have not been a problem with Nginx (if I ever get to this point), as the main process would fire up a new worker, in case of MySQL this resulted in a complete shutdown.

Obviously in case of MySQL this means the exploitation has failed and we have to wait until someone restarts the service. Next time we had better start scanning the PLT from the end of it.

Finding write()

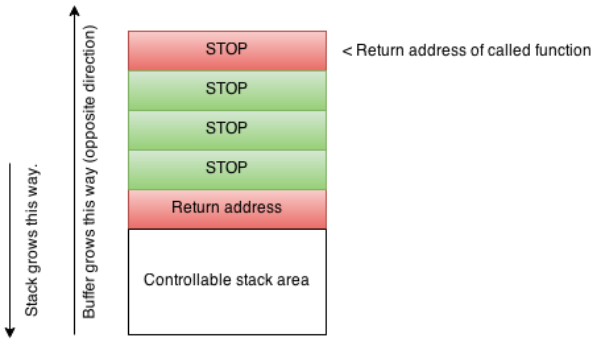
Finding `write()` is a bit more problematic than locating `strcmp()`. First, in case of MySQL the buffer size was too small to hold the payload I planned to use. Second, after overcoming the buffer size limitation, `write()` corrupted the binary log files which resulted in a complete service shutdown.

Extending the Buffer

To make our ROP chain fit in the buffer, we can try to further optimise the payload. Even if the payload is heavily optimised, it is still possible that it will not fit in the buffer.

If we take a look at Figure 2, again we will see that it should be possible to get some more bytes for our ROP payload. It should be possible to make function C overwrite function B's return address, or its own.

For this the previously controlled return address on the picture on the right had to be replaced with a trap, then the buffer had to be overflown further with stop gadgets. Once the return address of function B or function C got overwritten with the stop gadget, the service went into an infinite loop.



Why wasn't this the first return address to be found?

- In general this can happen either because the code probing for the buffer size started the overflow with a shorter value and incremented the overflow string until the first crash, which marked the first controllable return address, or because the code started between the two return addresses and as the first was overwritten with garbage, there was a crash, so the code started to decrement the size of the buffer to find where the return address started.
- In this specific case, if I set the overflow string to be really long I have overwritten all return addresses with junk and there is no way (remotely, with a blind attack) to figure out which function crashed. This is why the use of a stop gadget was required, as using it made it possible to check whether a value further down the stack could be a controllable return address.

Issues with Detection

Once we have a big enough buffer space to inject the `write()` lookup ROP chain, we can start brute-forcing the location of `write()`. It turned out this process is not completely reliable either, as both `pwrite()`, which was found first, and `write()` completely crashed MySQL and, as the binary log files got corrupted, it prevented restarting MySQL. The problem was that one of the probed file descriptors was the file descriptor of the binary log.

As stdin, stdout, and stderr are a no-go by default (descriptors 0, 1, 2), chances are that the next few descriptors are related to files MySQL working with. What we can do is to start with a higher number when brute-forcing the socket ID. The question is: where should we start brute-forcing from to avoid corrupting the binary logs but not to miss the right socket descriptor? Maybe this is one of the reasons the original BROP code opened so many connections to the target: to be able to start with a bigger socket ID and not to miss the correct one(s).

The other problem was that `pwrite()` was hit first: `pwrite()` has four arguments, and the last one is *offset* which we cannot control. The register `RCX` holding the value of the fourth argument can hold a value which might trigger an error, and as a result `pwrite()` will never return anything on the socket. We can expect the following problems with `pwrite()`, according to the manual page:

- "The file referenced by fd must be capable of seeking."
  A socket is not capable of seeking.

- "pwrite() can fail and set *errno* to any error specified for write() or lseek()."

If we have a look at the manual of lseek() we will see that lseek(), and therefore pwrite(), can fail with ENXIO, which means the file offset is beyond the end of the file.

The solution was the same as in case of the `strcmp()` lookup: scanning the PLT starting from the end. As we are now able to issue a `write()` call using the MySQL socket connection, it becomes possible to leak data from arbitrary addresses from the service.

Crafting the Final Payload

Once we are ready to dump the in-memory image of the binary, we can either extract useful information on the fly or dump the binary and parse it later. We can extract all the information required to create our final payload. I decided to dump a big part of the binary first.

Just as a side-note: At this point we are basically done with the "Blind" part of the ROP exploitation process. At first I thought from this point forward everything would be easy, but it turned out to be quite challenging and interesting, so I decided to write about these things as well.

Dumping the binary made finding ROP gadgets such as `POP <REG>; RET` trivial. One thing to note here is that the dumped binary is not identical to the original file before it got loaded into the memory. The section header, for example, is missing. This means that some ROP gadget finder tools might not be able to work with the dump file. This was the case, for example, with Ropper.

By enumerating the PLT entries and their relocation offsets it becomes possible to call any of the functions. But why would we bother enumerating PLT instead of finding a `syscall` gadget? One of the reasons is that when I have tried to look for a `syscall` gadget, I could not find any which would not result in a crash. This does not mean we cannot use it. It means we can use it only once.

The question is, how is it possible to figure out the relocation offset of the PLT entries, and how do we get their matching symbol names?

Finding the String Table

Fortunately there is not much explanation needed here. A string table is an array of null terminated strings. What we have to know is that the first byte of the string table is a NULL byte. Therefore, the string table starts with that initial null byte. This is important as all the offsets in the symbol table point to strings relative to this location. We start searching from the beginning of the dump until a list of strings is found. The following regexp code (Python) can be used to check for a match.

```
m = re.search("\0" + ("[0-9a-zA-Z\@\_\-\.]{1,512}\0" * 4), self.s) # self.s is
mmap'ed dump file
```

Finding the Symbol Table

To be able to prepare a good pattern, first we have to have a look at the structure of the symbol table. This can be seen below.

```
    typedef struct
    {
        Elf64_Word       st_name;     /* 4 bytes offset - symbol name */
        unsigned char    st_info;     /* 1 byte - type and Binding attributes */
        unsigned char    st_other;    /* 1 byte - reserved */
        Elf64_Half       st_shndx;    /* 2 bytes - section table index */
        Elf64_Addr       st_value;    /* 8 bytes - symbol value */
        Elf64_Xword      st_size;     /* 8 bytes - size of object (e.g., common)
*/
    } Elf64_Sym;
```

Based on the above we know that a symbol table entry is twenty-four bytes long. What we additionally have to know is that the first symbol table entry is reserved and must be all zeroes. The symbolic constant `STN_UNDEF` is used to refer to this entry.

What I have implemented is extremely simple and assumes the symbol table follows the string table. I scan from the string table towards higher offsets until twenty-four bytes of 0x00 is found. While this works fine for me, one might extend the lookup pattern by looking for specific `st_info` field values (type and binding attributes) to reduce false positives.

Finding the Relocation Table

To be able to prepare a good pattern, first we have to have a look at the structure of the relocation table. This can be seen below.

```
typedef struct
{
    Elf64_Addr     r_offset;   /* Address of reference */
    Elf64_Xword    r_info;     /* Symbol index and type of relocation */
    Elf64_Sxword   r_addend;   /* Constant part of expression */
} Elf64_Rela;
```

An entry in the relocation table is twenty-four bytes long. The `r_info` field holds the symbol index and the type of the relocation, and is of use to us. The relocation types for x64 can be seen below. (Taken from `/usr/src/linux-headers-3.16.0-30/arch/x86/include/asm/elf.h`)

```
/* x86-64 relocation types */
#define R_X86_64_NONE           0       /* No reloc */
#define R_X86_64_64             1       /* Direct 64 bit  */
#define R_X86_64_PC32           2       /* PC relative 32 bit signed */
#define R_X86_64_GOT32          3       /* 32 bit GOT entry */
#define R_X86_64_PLT32          4       /* 32 bit PLT address */
#define R_X86_64_COPY           5       /* Copy symbol at runtime */
#define R_X86_64_GLOB_DAT       6       /* Create GOT entry */
#define R_X86_64_JUMP_SLOT      7       /* Create PLT entry */
#define R_X86_64_RELATIVE       8       /* Adjust by program base */
#define R_X86_64_GOTPCREL       9       /* 32 bit signed pc relative
                                           offset to GOT */
#define R_X86_64_32             10      /* Direct 32 bit zero extended */
#define R_X86_64_32S            11      /* Direct 32 bit sign extended */
#define R_X86_64_16             12      /* Direct 16 bit zero extended */
#define R_X86_64_PC16           13      /* 16 bit sign extended pc relative */
#define R_X86_64_8              14      /* Direct 8 bit sign extended  */
#define R_X86_64_PC8            15      /* 8 bit sign extended pc relative */
#define R_X86_64_NUM            16
```

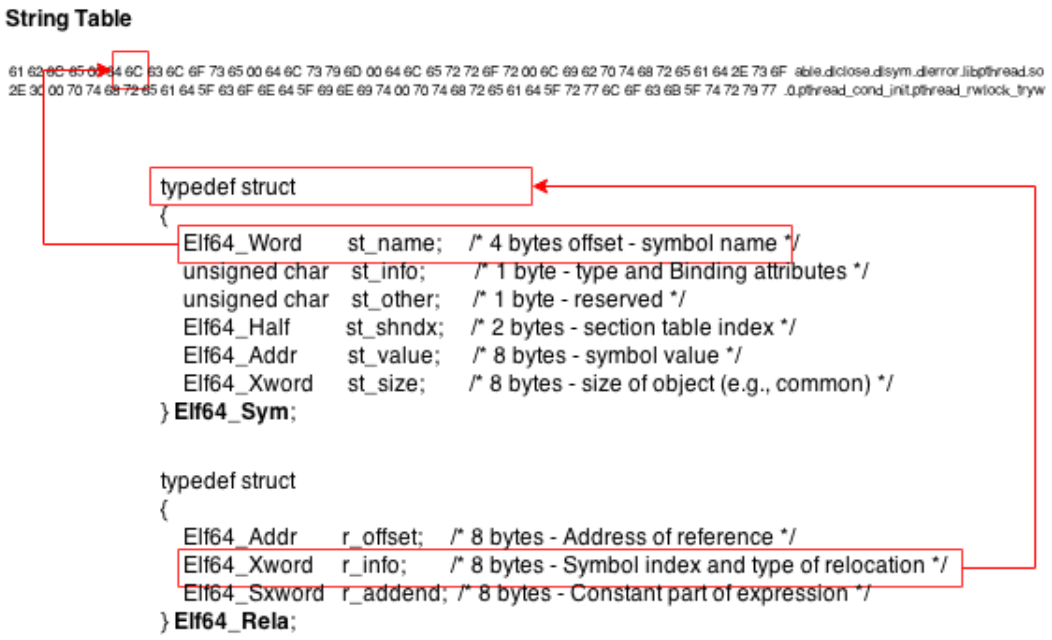
We are interested in the `R_X86_64_JUMP_SLOT` relocation type (there is a reason why I marked the `R_X86_64_COPY` relocation type, but more on this later). So we can start scanning from the string table towards higher offsets to find a twenty-four byte entry where the extracted and parsed `r_info` field will yield this relocation type. Additionally, `r_addend` would be 0x00 should we also check for this.

The following Python snippet shows this:

```
    entry    = self.f.read(24)
    r_info_type = struct.unpack("<I", entry[8:12])[0]
    r_addend = struct.unpack("<Q", entry[16:24])[0]
    if r_info_type == 0x7 and r_addend == 0x0: return True
```

Parsing the Relocation Table

The following picture explains how to parse the relocation table, look up the symbol table entry belonging to the relocation, and then read the name of the symbol from the string table.



Once we have the symbol-name - relocation pairs we will know which relocation offset belongs to the functions. We can use the relocation offsets to call the functions from the PLT using the PLT gadget found earlier.

And now comes the interesting part: I had trouble resolving symbol names of the relocations properly. Related documents only mention that the relocation offset extracted from `r_info` using the `ELF64_R_SYM` macro points to a symbol entry in the symbol table. For example `R_SYM` 3 points to the third item in the symbol table. But when I tried to map the relocation table entries to the symbol table entries based on the extracted symbol index I got incorrect results. I spotted this because I had several reference points within the PLT such as `strncmp()` and `write()` with the correct relocation offsets obtained by brute-forcing the PLT earlier.

As the relocation table did not reference symbol entry #0 and the first relocation entry referenced #1, it turned out that I have to subtract 1 from the `R_SYM`. I also found that sometimes a relocation entry is “missing” from the table, as for example an entry with relocation offset 0x21 was followed by 0x23. It turned out that if a relocation offset is “missing”, the next one should be moved into its place and its relocation offset should be updated, but the symbol table lookup should still point to the correct value as it seems to be independent from the relocation table issues. This manifested the following ugly Python solution.

```
    r_sym = (r_info >> 32) - 1
    if r_sym != CNT:
```

```
        r_sym -= 1
        s_sym = self.sym_lookup(r_sym + 1)
        if r_sym > CNT + 1: r_sym = CNT
    else:
        s_sym = self.sym_lookup(r_sym)
```

Please note that the `CNT` variable above was used to keep track of the real index of the relocations in the relocation table.

At this point we can call any function from the PLT and extract useful gadgets from the binary. Next, we have to deal with the small buffer which is not big enough to hold a complex payload, as mentioned earlier in the MySQL case.

Extending the Buffer, Again

Depending on the size of the buffer we have several options, which include:

- Using a `SUB ESP, ???; RET` gadget to extend the stack area and align our payload in a way that, as a result of the gadget `RSP,` will point to our ROP chain.
- Use `mmap()` to map an executable region with `PROT_EXEC` flag, call `read()` to read a second-stage payload (does not have to be a ROP chain) to the area, and get it executed. Without `PROT_EXEC`, `RSP` has to be set to the mapped are and a ROP chain has to be used.

In case of MySQL, the stack redirection approach was followed with some important differences, because calling `mmap()` itself would not have fit on the stack. I had no suitable `SUB ESP, ???; RET` gadget anywhere. Neither `ADD ESP` with a negative value. BUT, I had a relocation table.

The idea was to find an entry in the relocation table with the type `R_X86_64_COPY`. The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset. Once the entry found the `r_offset` value, which points to writable memory area, it can be unpacked, aligned by subtracting the payload size from it (e.g. 0x800) which would allow 2048 bytes, a total of 256 entries on the stack to be used for a second stage ROP chain. Later, an ROP chain can be created which uses `write` to write data read from the socket into the memory area. As the last step, the stack pointer can be redirected using a `POP RSP; RET` instruction.

I have used this newly "allocated" area to host a ROP payload which maps an executable memory region and reads a third-stage payload into it, then passes execution to it.

Conclusions and Summary

BROP is a really time-consuming exploitation technique. From my experience, it works much better in theory than in practice. Successful exploitation depends on several factors, such as when, by what compiler, and how, the target application was compiled, what protections were applied, how the application works, how robust the error handling implemented is, how it behaves when a thread or worker crashes, luck, and so on.

Still, BROP is something extremely fun and challenging to play with. I hope you enjoyed reading this post as much as I enjoyed writing the exploit.