Lab3: Stack Smashing

1 Introduction

Software inevitably have bugs, and taking advantage of such bugs to control program behavior is called *software exploitation*. This lab tests your knowledge on how computers work through a set of software attack challenges. You are given four executables that have classical *stack overflow* vulnerabilities. Your task is to put everything you learned so far into practice and write attack code that will allow you to hijack the program control flow.

2 Handout Organization and Deliverables

Similar to Lab2, you will be given an archive file, and you can decompress the file with the following command:

```
$ tar -zxvf 20xxxxxxxx.tar.gz
```

You will see the following files in the decompressed directory:

```
./20xxxxxxxx/
├── advanced-example.py // gdb-pwntools integration example
├── install-deps.sh // installs python and pwntools
├── stage1
├── stage1.py
├── stage2
├── stage2.py
├── stage3
├── stage3.py
├── stage4
└── stage4.py
```

Executables and exploit scripts. `stage1`~`stage4` are the executable binaries with stack overflow vulnerabilities. `stage1.py` ~`stage4.py` are the exploit skeleton files to which you will write your attack code.

Stage pass condition. Your goal in each stage is to control the vulnerable programs to print the following message:

```
##########################################################
ACCESS GRANTED: You have passed StageN
##########################################################
```

Note that each executable already has code that prints the above message. Hence, you will satisfy the conditions for the message to print (`stage1`) or hijack the program control flow through `ROP` attacks (`stage2` ~`stage4`) (hint: take a look at `escape()`). Also, you will see that each executable has devises that makes the challenge easier to solve. Make sure you examine all symbols with interesting names. For example, the following command in shell prompt will print all the symbols of an executable.

```
$ nm stage1
```

Deliverables. You will write attack code in python using *pwn tools*. Your final deliverables will be the python code for each of the stages `stage1.py` ~`stage4.py`.

Score distribution.

Task	Description	Points
Task 1	Password Prompt Bypassing	20 / 100 pts
Task 2	Simple `retAddr` Overwrite	20 / 100 pts
Task 3	Easy ROP Room Escape	30 / 100 pts
Task 4	Hard ROP Room Escape	30 / 100 pts

3 Submission

You will submit `2021xxxxxx` directory with the `.py` files completed, zipped with `tar` (e.g., `2021xxxxxx.tar.gz`) to *icampus*.

Archiving the handout directory. Be warned that our grading system would not recognize other compression formats such as `.zip` or `.7z`. Hence, we ask you to archive the directory exactly as the follows:

```
./2021xxxxxx
```

Academic integrity. This is an individual lab, and you are *not allowed* to discuss with your classmates. Your keygen program will be checked for plagiarism using our system. Be warned that it is extremely difficult to fool the plagiarism checker, and it is wise not to copy someone

else's work in the first place. Any identified act of academic dishonesty would result in a Lab3 grade of zero.

4 Quick Starter Guide

4.1 GDB

I highly recommend using GDB with a plugin such as **`https://github.com/hugsy/gef`** that provides a better visualization and additional commands. Many tutorial on using GDB is available on the internet. Therefore, here we will highlight the main functionalities of GDB that are necessary for doing this lab.

AT&T vs. Intel notation for x86 assembly. In this course, we learned the x86 assembly language using the AT&T notation. Depending on the plugin you install, Intel notation may be set by default. You can change this back to AT%T by typing "**`set disassembly-flavor att`**" inside GDB.

Disassembling code. As you may have learned through the previous lab, you can disassemble a function or an address with the **`disas`** command. The following is the disasembly of a function called **`overflow`**

```
gef➤  disas overflow
Dump of assembler code for function overflow:
0x000000000040124a <+0>:     push   %rbp
0x000000000040124b <+1>:     push   %rbx
0x000000000040124c <+2>:     sub    $0x28,%rsp
...
```

Figuring out the stack. There are a few indications that allow you to picture the justxapositionof the function's stack frame. Remember that **`%rsp`** points to the saved return address when entered a newly invoked function. Then, the function's stack frame is built on top of that. For instance, in the above disassembled code, there are two **`push`** instructions which will decrement **`%rsp`** by 8 bytes * 2. Subsequently, a **`sub $0x28, %rsp`** further decrement the stack by 40 bytes which gives us a 40 bytes of scratch space we can use for local variables. Also you should also note that we are now 56 bytes away from the saved returned address which is our attack target. Another bit of information that you are interested in is the exact location of a stack variable with respect to **`%rsp`**. You can easily earn this information by looking at the function parameters. For instance, imagine that you would like know the

exact location of `char array[16];`. First findthe operations or functions that use that variable (e.g., `printf("%s",array);`). Then take a look at its parameters (`%rdi,%rsi,...`), and you'll usually see that the variable is referenced with respect to the stack pointer (e.g., `%rsp+N`).

Runtime debugging. When you're stuck, an online debugging can be of help. When you have opened the executable with GDB, you can type the `start` (provided by gef) command to put the program into a state where it's just about to start `main`. You can also set breakpoints at arbitrary points (e.g., `b *my_function+19`, `b *0xdeadbeef`. The program will stop at the breakpoint, and gdb will show you the visualization of the current program state.

4.2 Pwntools

Pwntools is a python library for writing software exploits, and is widely used in real CTFs (Hacking competition) and real-world attacks. A comprehensive documentation is available at `https://docs.pwntools.com/en/latest/`. But we will only use a small subset of the features for this lab. Also, most of the code below are provided in the skeleton files.

Opening executables. below code shows how to open an executable.

```
import sys
from pwn import process, context, p64, ELF
import re

# Get target executable
target = sys.argv[1]
context.binary = ELF(target, checksec=False)
```

Reading addresses of symbols. at this point, addresses of symbols (e.g., functions and global variables) can be easily parsed from the executable as the following:

```
my_function = context.binary.symbols['my_function']
my_variable = context.binary.symbols['my_variable']
```

Launching and communicating with processes. You can launch the program from within python, and receive program outputs and also pass input to the program. This enables us (attackers) to inject inputs (e.g.,) that are not possible to generate with your keyboard.

```
p = process(target)
io = p
```

```
# Receive output from program and print it in ASCII
# This will get us to the prompt shown by the program
output = io.recv()
print(output.decode('ascii'))

# add 16 'a's
input = b'a'*16
# p64 'packs' 64 bit integer into little-endian bytes
input += p64(my_function)

# send and receive program output
io.sendline(input)
output = io.recv()
print(output.decode('ascii'))
```

The above program launches a process of the exectuable, then receives its initial output. The goal here is to proceed to the program state where the program is waiting for your input (e.g., `Enter the your attack code >>`). Then, we craft a input value that will fill the stack. The example fills the input with 16 `'a'`s, then a address of `my_function`. `p64` conveniently converts an integer into a little-endian byte sequence. In Line 15-17, we enter the attack code into the program, then get the program output. If you have successfully exploited the program, perhaps by overwriting the return address and redirecting program control flow to your favor, you'll see the ACCESS GRANTED message.

Advanced features. pwn tools provides additional functionalities that can be leveraged to make fully automated exploits in many cases. For instance, one can directly acquire the disassembled code through `context.binary.disasm(fn, n)`. Also, you can control GDB using python, and perform a live analysis of your exploit (`https://docs.pwntools.com/en/stable/gdb.html`). While these features may be an overkill for this lab, I had to write a fully automated exploits to test all your executables, since everyone gets a slightly different executables :)

And Lastly... Have Fun!!!