Lab06

Buffer Overflow and Shell Code

[You must not attack any network without authorization! There are also severe legal consequences for unauthorized interception of network data.]

Introduction

This project will introduce you to control-flow hijacking vulnerabilities in application software, including buffer overflows. You will be working through this in a virtual machine environment starting with some practice programs for you to get familiar with the tools you need. We will then provide a series of vulnerable programs which you will develop exploits.

Objectives

Be able to identify and avoid buffer overflow vulnerabilities in native code.

• Understand the severity of buffer overflows and the necessity of standard defenses.

• Gain familiarity with machine architecture and assembly language.

Setup

Buffer-overflow exploitation depends on specific details of the target system, so we are providing an Ubuntu VM in which you should develop and test your attacks. We've also slightly tweaked the configuration to disable security features that would complicate your work. We'll use this precise configuration to grade your submissions, so you MUST NOT use your own VM.

1. Download VirtualBox from https://www.virtualbox.org/ and install it on your computer. VirtualBox runs on Windows, Linux, and Mac OS.

2. Get the VM file that we gave you at the lab. This file is 1.3 GB, so we recommend taking this on your pen drive.

3. Launch VirtualBox and select Import Appliance to add the VM.

4. Start the VM. There is a user named ubuntu with password ubuntu.

5. You will also need **this file** to complete your lab work.
6. untar the files and change to that directory. Run ./setcookie firstname . make

Resources and Guidelines

No Attack Tools! You MUST NOT use special-purpose tools meant for testing security or exploiting vulnerabilities. You MUST complete the project using only general purpose tools, such as gdb. You will make extensive use of the GDB debugger. Useful commands that you may not know are “disassemble”, “info reg”, “x”, and setting breakpoints. See the GDB help for details, and don’t be afraid to experiment! This quick reference may also be useful:

http://www.tenouk.com/Module000linuxgcc1.html

x86 Assembly These are many good references for Intel assembly language, but note that this project targets the 32-bit x86 ISA. The stack is organized differently in x86 and x86_64. If you are reading any online documentation, ensure that it is based on the x86 architecture, not x86_64.

10.1.1 GDB practice (4 points)

This program really doesn’t do anything on its own, but it allows you to practice GDB and look at what the program is doing at a lower level. You have two jobs here. The first job is to look at where the function practice is going to return. The second job is to find out what’s the value in register eax right before the function practice returns.

Here’s one approach you might take:

1. Start the debugger `(gdb 4.1.1),` set a breakpoint at function practice:
`(gdb) b practice`, then run the program: `(gdb) r.`

2. Think about where the return address would be relative to register ebp (the base pointer).

3. Examine`(x in gdb)` that memory location:
`(gdb) x 0xAddress or (gdb) x $ebp+#` and put your answer in 4.1.1_addr.txt.

Remember you can use `(gdb) info reg` to look at the values of registers at that breakpoint!

4. Disassemble `practice` with `(gdb) disas practice` then set a breakpoint at the address of ret instruction to pause the program right before `practice` returns. You can do this with `(gdb) b *0x0804dead` if the ret instruction is located at address `0x0804dead`. After that, continue to that breakpoint: `(gdb) c.`

5. With `(gdb) info reg`, you can look at the value in `eax` and put your answer in 4.1.1_eax.txt.

Submit the return address of the function practice in 4.1.1_addr.txt and the value of eax right before practice returns in 4.1.1_eax.txt

10.1.2 Assembly practice (4 points)

This time, the function `practice` prints different things depending on the arguments. Your job is to call the C function from your x86 assembly code with the correct arguments so that the C function prints out `"Good job!".` Here are some questions to think about (you do not need to submit the answers to these):

1. How are argruments passed to a C function?
2. In what order should the arguments be pushed onto the stack?

Tips:
1. Use `push $0x12341234` to push arbitrary hex value onto the stack.

2. Use `call function_name` to call functions.

Submit your x86 assembly code in 4.1.2.S. Make sure the entire program exits properly with your assembly code!

10.1.3 Assembly practice with pointer(s) (4 points)

Just like 4.1.2, your goal is to call the function practice with the correct arguments so that the function prints out "Good job!". Notice that the parameters are slightly different than 4.1.2. Hint: Think about what would be on top of the stack if you do:

```
push $0x12341234
mov %esp,%eax
push %eax
```

Submit your x86 assembly code in 4.1.3.S. Make sure the entire program exits properly with your assembly code!

10.1.4 Assembly practice with pointer(s) and string(s) (4 points)

Just like 4.1.2 and 4.1.3, your goal is to call the function practice with the correct arguments so that the function prints out "Good job!". Notice that the parameters are slightly different than 4.1.2 and 4.1.3.

Tips:
1. Byte order for x86 is little endian

2. Characters are read from top to bottom of the stack (low memory to high memory)

3. What character/value indicates end of string?

Submit your x86 assembly code in 4.1.4.S. Make sure the entire program exits properly with your assembly code!

10.1.5 Introduction to Linux function calls (4 points)

Your goal for this practice is to invoke a system call through int 0x80 to open up a shell.
Tips:

1. Use the system call `sys_execve` with the correct arguments.

2. The funtion signature of `sys_execve` in C:
```
int execve(const char *filename, char *const argv[],char *const envp[]);
```

3. Instead of passing the arguments through the stack, arguments should be put into registers for system calls.

4. The system call number should be placed in register `eax`.

5. The arguments for system calls should be placed in ebx, ecx, edx, esi, edi, and ebp in order

6. To start a shell, the first argument (filename) should be a string that contains something like `/bin/sh.`
7. Reading Linux man pages may help.

8. Some arguments may need to be terminated with a null character/pointer.

Submit your x86 assembly code in 4.1.5.S

10.2.1 Overwriting a variable on the stack (8 points)

This program takes input from stdin and prints a message. Your job is to provide input that makes it output: "Hi *yourname* ! Your grade is A+.". To accomplish this, your input will need to overwrite another variable stored on the stack.
Here's one approach you might take:

1. Examine 4.2.1.c. Where is the buffer overflow?

2. Start the debugger (gdb 4.2.1) and disassemble _main: `(gdb) disas _main`
Identify the function calls and the arguments passed to them.

3. Draw a picture of the stack. How are `name[]` and `grade[]` stored relative to each other?

4. How could a value read into `name[]` affect the value contained in `grade[]`? Test your hypothesis by running ./4.2.1 on the command line with different inputs.

What to submit Create a Python program named 4.2.1.py that prints a line to be passed as input to the target. Test your program with the command line:
python 4.2.1.py | ./4.2.1

Hint: In Python, you can write strings containing non-printable ASCII characters by using the escape sequence "`\xnn`", where nn is a 2-digit hex value. To cause Python to repeat a character n times, you can do: print "X"*n.

10.2.2 Overwriting the return address (8 points) (Difficulty: Easy)

This program takes input from stdin and prints a message. Your job is to provide input that makes it output: "`Your grade is perfect`." Your input will need to overwrite the return address so that the function `vulnerable()` transfers control to `print_good_grade()` when it returns.

1. Examine 4.2.2.c. Where is the buffer overflow?

2. Disassemble `print_good_grade`. What is its starting address?

3. Set a breakpoint at the beginning of vulnerable and run the program.
```
(gdb) break vulnerable
(gdb) run
```

4. Disassemble vulnerable and draw the stack. Where is `input[]` stored relative to %ebp? How long an input would overwrite this value and the return address?

5. Examine the `%esp and %ebp registers: (gdb) info reg`

6. What are the current values of the saved frame pointer and return address from the stack frame? You can examine two words of memory at `%ebp` using: `(gdb) x/2wx $ebp`

7. What should these values be in order to redirect control to the desired function?

What to submit Create a Python program named 4.2.2.py that prints a line to be passed as input to the target. Test your program with the command line:
python 4.2.2.py | ./4.2.2

When debugging your program, it may be helpful to view a hex dump of the output. Try this:
python 4.2.2.py | hd

Remember that x86 is little endian. Use Python's struct module to output little-endian values:

```
from struct import pack
print pack("<I", 0xDEADBEEF)
```

10.2.3 Redirecting control to shellcode (8 points)

The remaining targets are owned by the root user and have the suid bit set. Your goal is to cause them to launch a shell, which will therefore have root privileges. This and later targets all take input as command-line arguments rather than from stdin. Unless otherwise noted, you should use the shellcode we have provided in shellcode.py. Successfully placing this shellcode in memory and setting the instruction pointer to the beginning of the `shellcode` (e.g., by returning or jumping to it) will open a shell.

1. Examine 4.2.3.c. Where is the buffer overflow?

2. Create a Python program named 4.2.3.py that outputs the provided shellcode:

```
from shellcode import shellcode
print shellcode
```

3. Set up the target in GDB using the output of your program as its argument:
`gdb --args ./4.2.3 $(python 4.2.3.py)`

4. Set a breakpoint in vulnerable and start the target.

5. Disassemble vulnerable. Where does buf begin relative to %ebp? What's the current value of %ebp? What will be the starting address of the shellcode?

6. Identify the address after the call to strcpy and set a breakpoint there:
`(gdb) break *0x08048efb`
Continue the program until it reaches that breakpoint.
(gdb) cont

7. Examine the bytes of memory where you think the shellcode is to confirm your calculation:
`(gdb) x/32bx 0xaddress`

8. Disassemble the shellcode: `(gdb) disas/r 0xaddress,+32`
How does it work?

9. Modify your solution to overwrite the return address and cause it to jump to the beginning of the shellcode.

Create a Python program named 4.2.3.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:
./4.2.3 $(python 4.2.3.py)

If you are successful, you will see a root shell prompt (#). Running `whoami` will output "root". If your program segfaults, you can examine the state at the time of the crash using GDB with the core dump: `gdb ./4.2.3 core`. The file core won't be created if a file with the same name already exists. Also, since the target runs as root, you will need to run it using `sudo ./4.2.3` in order for the core dump to be created.