Project 4: Application Security

This project is due on **Thursday, April 21** at **11:59pm**. This is a group project: you can work in teams of two and can share the same answers. If you are having trouble forming a teammate, please post to the #partner-search channel on Slack.

Your answers here must be entirely your own work and that of your partner. You may discuss the problems with other students outside your group, but you may not directly copy answers, or parts of answers.

Introduction

This project will introduce you to control-flow hijacking vulnerabilities in compiled software, specifically buffer overflows. We will provide a series of vulnerable programs which you will need to hijack.

This project asks you to develop attacks and test them in a virtual machine you control. Like always, attempting the same kinds of attacks against others' systems without authorization is prohibited by law and university policies and may result in fines, expulsion, and jail time. You must not attack anyone else's system without authorization!

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 disabled some security features that are commonly used in the wild but would complicate your work. We'll use this precise configuration to grade your submissions, so do not use your own VM instead!

1. Download VirtualBox from https://www.virtualbox.org/ and install it on your computer.. VirtualBox runs on Windows, Linux, and Mac OS.
2. Download the app_sec_vm.ova VM file from Canvas.
3. Launch VirtualBox and select "File" >"Import Appliance" to add the VM.
4. Start the VM. A username and password are not required to login, but if needed they are **ubuntu** and **ubuntu**.
5. Download the targets onto the VM. They are hosted on GitHub, so you can retrieve them with the command
   ```
   $ git clone https://github.com/apccurtiss/lab_4_buffer.git
   ```
   You can also log into Canvas and download the lab_4_buffer.zip file from there.
6. Compile the targets by running:
   ```
   $ sudo make
   ```

The root password is **ubuntu**

Resources and Guidelines

No Attack Tools! You may not use special-purpose tools meant for testing security or exploiting vulnerabilities. You must complete the project using only general purpose tools, such as Ghidra and GDB.

You will make extensive use of Ghidra and GDB. We have worked with Ghidra in recitation, and you should be passingly familiar with GDB. Some useful GDB commands will be “disassemble”, “info reg”, “x”, and “stepi”. See man gdb for more details, and don’t be afraid to experiment! Installing Ghidra in the VM is a pain, so we suggest you also download the files locally and open them in Ghidra there.

Targets

The target programs for this project are simple, short C programs with (mostly) clear security vulnerabilities. We have provided source code and a makefile that compiles all the targets. Your exploits must work against the targets as compiled and executed within the provided VM.

target0: Overwriting a variable on the stack (25 points)

This program takes input from stdin and prints a message. Your job is to provide input that causes the program to output: “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 target0.c. Where is the buffer overflow?
2. Analyze the file in Ghidra. How are name[] and grade[] stored relative to each other? Drawing a picture of the stack may help.
3. How could a value read into name[] affect the value contained in grade[]? Test your hypothesis by running ./target0 on the command line with different inputs.

What to submit: Create a Python 2 program named sol0.py that prints a line to be passed as input to the target. Test your program with the command line:

```
python sol0.py | ./target0
```

Note: We have used Python 3 before now, but Python 3 handles strings in a much less intuitive way. We will use Python 2 for this one. The VM has Python 2 installed by default.

In Python 2, 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. So "\x90" * 4 will print 4 bytes equal to the hex number 0x90909090.

target1: Overwriting the return address (25 points)

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.

Remember that x86 is little endian, so you will need to flip the order of bytes of each value you inject into the stack! You can turn an integer into a little-endian string in Python with the following code:

```
from struct import pack
pack("<I", 0x12345678) # returns '\x78\x56\x34\x12'
```

Your process could go like this:

1. Examine target1.c. Where is the buffer overflow?
2. Disassemble the function print_good_grade. What is its starting address?
3. Disassemble vulnerable and draw the stack like before. Where is input[] stored relative to the return address? How long would an input have to be to overwrite that address?

At this point, you should have enough information to perform your attack. However, you may want to debug your attack by viewing the registers during runtime:

4. Set a breakpoint at the beginning of vulnerable and run the program.
   ```
   (gdb) break vulnerable
   (gdb) run
   ```
5. Examine the %esp and %ebp registers:
   ```
   (gdb) info reg esp
   (gdb) info reg ebp
   ```
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 starting at %ebp:
   ```
   (gdb) x/2wx $ebp
   ```
7. What do these values point to now? What should they be in order to redirect control to the desired function?

What to submit: Create a Python 2 program named sol1.py that prints a line to be passed as input to the target. Test your program with the command line:

```
python sol1.py | ./target1
```

When debugging your program, it may be helpful to view a hex dump of the output. You can use:

```
python sol1.py | hd
```

target2: Redirecting control to shellcode (25 points)

For the remaining targets, your goal is to cause them to launch a shell which will let you run additional commands. This and 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 target2.c. Where is the buffer overflow?
2. Create a Python program named sol2.py that outputs the provided shellcode:
   ```
   from shellcode import shellcode
   print(shellcode)
   ```
3. Because the stack addresses are not calculated until runtime, we will need to use GDB to determine where our shellcode will be placed. Load the target in GDB using the output of your program as its argument:
   ```
   gdb --args ./target2 $(python sol2.py)
   ```
4. Set a breakpoint in *vulnerable* and start the target.
5. Disassemble *vulnerable*. Where does the buffer begin relative to %ebp? What's the current value of %ebp? What will be the starting address of the shellcode you injected?
6. Identify the address after the call to strcpy and set a breakpoint there:
   ```
   (gdb) break *0xaddress
   ```
7. Continue the program until it reaches that breakpoint.
   ```
   (gdb) continue
   ```
8. Examine the bytes of memory where you think the shellcode is to confirm your calculation:
   ```
   (gdb) x/64bx 0xaddress
   ```
9. Modify your solution to overwrite the return address and cause it to jump to the beginning of the shellcode.

What to submit: Create a Python program named sol2.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./target2 "$(python sol2.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 ./target2 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 ./target2 in order for the core dump to be created.

target3: Bypassing DEP (25 points)

This program resembles target2, but it has been compiled with data execution prevention (DEP) enabled. DEP means that the processor will refuse to execute instructions stored on the stack. You can overflow the stack and modify values like the return address, but you can't jump to any shellcode you inject. You need to find another way to run the command /bin/sh and open a root shell.

What to submit: Create a Python program named sol3.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./target3 "$(python sol3.py)"
```

For this target, it's acceptable if the program segfaults after the root shell is closed.

target4: Return-oriented programming [Extra credit] (25 points)

This target is identical to target2, but it is compiled with DEP enabled. Implement a ROP-based attack to bypass DEP and open a root shell.

What to submit: Create a Python program named sol4.py that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./target4 "$(python sol4.py)"
```

You may find the ROPgadget tool helpful (https://github.com/JonathanSalwan/ROPgadget) as well as the objdump command.

Submission Checklist

Please upload the following answers to Canvas in a single .zip file. The name of the zip file should include your identikey and your partner's identikey (if you have a partner). **Each member of your team should upload the .zip file to get graded**. Your submission should contain the following files. **Make sure you follow the naming conventions described below**, since portions of the assignment will be autograded!

sol0.py
sol1.py
sol2.py
sol3.py
sol4.py [Optional extra credit]

Your files can make use of standard Python libraries and the provided shellcode.py, but they must be otherwise self-contained. Be sure to test that your solutions work correctly in the provided VM without installing any additional packages.