Project 1: Application Security

This project is due on **Thursday, September 19, 2019** at **10 PM**. **You must work in teams of two** and submit one project per team. You will have a budget of five late days (24-hour periods) over the course of the semester that you can use to turn assignments in late without penalty and without needing to ask for an extension. You may use a maximum of two late days per assignment. Late pair projects will be charged to both partners. Once your late days are used up, extensions will only be granted in extraordinary circumstances.

The code and other answers your group submits must be entirely your own work, and you must adhere to the Code of Academic Integrity. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically **via Canvas**, following the submission checklist at the end of this file.

Introduction

This project will introduce you to control-flow hijacking vulnerabilities in application software, including buffer overflows. We will provide a series of vulnerable programs and a virtual machine environment in 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.

Read this First

This project asks you to develop attacks and test them in a virtual machine you control. 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!** You must respect the privacy and property rights of others at all times.

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 have also slightly tweaked the configuration to disable security features that would complicate your work. We will 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, and is installed in the Moore 100 labs.
2. Get the VM image from https://www.cis.upenn.edu/~cis331/project1/cis331-proj1-vm.ova. This file is 1.6 GB, so we recommend downloading it from campus.
3. Launch VirtualBox and select File ▷ Import Appliance to add the VM.
4. Start the VM. There is a user named `ubuntu` with the password `ubuntu`.
5. Open the terminal and download the target programs that you will exploit:
   `wget` https://www.cis.upenn.edu/~cis331/project1/cis331-proj1-targets.tar.gz
6. `tar xf cis331-proj1-targets.tar.gz`
7. `cd targets`
8. Each group's targets will be slightly different. Personalize the target by running:
   `./setcookie` *`PennKey 1 PennKey 2`*
   Make sure both PennKeys are correct and represent both team member's PennKeys.
9. `sudo make`

Optional step. Note that depending on your setup, VirtualBox's GUI might be slow or the screen might have low resolution, making it hard to code. One thing you can do is you can ssh into your VM. To do this, we will need to forward some port (for example 3022) in the host (your regular OS) to port 22 (the port used by ssh) in the guest (the VM we have provided). You will also need to install an ssh server in the guest. Here's a step-by-step.

1. Open VirtualBox, select the CIS 331 App Security Project VM.
2. Click on Settings, go to Network, click Advanced, then Port Forwarding.
3. Click the add a new rule button (should be some kind of + sign).
4. Set the following values. Name: ssh. Protocol: TCP. Host IP: leave empty. Host Port: 3022. Guest IP: leave empty. Guest Port: 22.
5. Press Ok, then restart the VM.

6. Inside the VM, open terminal and run:
   `sudo apt-get update`.
   Don't run apt-get upgrade.

7. Install an ssh server: "`sudo apt-get install openssh-server`"

At this point you can ssh into your VM when it is running. You can open your host OS's terminal (on Windows you can use the Linux Subsystem's bash terminal, Cygwin, or PuTTY) and run:

`ssh -p 3022 ubuntu@127.0.0.1`.

Resources and Guidelines

Control Hijacking Before you begin this project, read "Smashing the Stack for Fun and Profit" available at https://cis.upenn.edu/~cis331/resources/stack-smashing.pdf.

GDB You will need to make extensive use of the GDB debugger. Useful commands that you may not know are "`disassemble`", "`info reg`", "`x`", and "`stepi`". See the built-in GDB help for details, and don't be afraid to experiment! This quick reference may also be useful: https://www.cis.upenn.edu/~cis331/resources/gdb-refcard.pdf

x86 Assembly Language There are many good online 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 the x86_64 architecture. Here is one reference to the syntax that we are using: https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s3

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

(*Difficulty: Easy, 3 Points*)

This program takes input from `stdin` and prints a message. Your job is to provide input that makes it output: "`Hi` *`PennKey`*`! Your grade is A+.`" (You can use either group member's PennKey.) 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`. Use the `man` command line tool if you do not know what a particular C library function or system call does (e.g., `man gets`). Where is the buffer overflow?
2. Start the debugger (`gdb target0`) 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 `./target0` on the command line with different inputs.

What to submit Create a simple Python program named `sol0.py` that prints a line to be passed as input to the target. If you do not know Python, the following suffices:

```
print("add the line you want to print here")
```

Test your program with the command line:

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

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

target1: Overwriting the return address

(*Difficulty: Easy, 3 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.

1. Examine `target1.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 `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. Try this:

```
python sol1.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)
```

target2: Redirecting control to shellcode

(*Difficulty: Easy, 3 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 `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. Set up 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 `buf` begin relative to `%ebp`? What is 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 0x`*`address`*

8. Disassemble the shellcode: `(gdb) disas/r 0x`*`address`*`,+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.

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: Overwriting the return address indirectly (*Difficulty: Medium, 4 Points*)

In this target, the buffer overflow is restricted and cannot directly overwrite the return address. You will need to find another way. Your input should cause the provided shellcode to execute 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)
```

target4: Beyond strings (*Difficulty: Medium, 4 Points*)

This target takes as its command-line argument the name of a data file it will read. The file format is a 32-bit count followed by that many 32-bit integers. Create a data file that causes the provided shellcode to execute and opens a root shell.

What to submit Create a Python program named `sol4.py` that outputs the contents of a data file to be read by the target. Test your program with the command line:

```
python sol4.py > tmp; ./target4 tmp
```

target5: Bypassing DEP

(*Difficulty: Medium, 4 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 `sol5.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./target5 $(python sol5.py)
```

For this target, it's acceptable if the program segfaults after the root shell is closed. Do not create a solution that depends on you manually setting environment variables. You cannot assume that we will run your solution with the same environment variables that you have set.

target6: Variable stack position

(*Difficulty: Medium*, 4 Points)

When we constructed the previous targets, we ensured that the stack would be in the same position every time the vulnerable function was called, but this is often not the case in real targets. In fact, a defense called ASLR (address-space layout randomization) makes buffer overflows harder to exploit by changing the position of the stack and other memory areas on each execution. This target resembles `target2`, but the stack position is randomly offset by 0–255 bytes each time it runs. You need to construct an input that always opens a root shell despite this randomization.

What to submit Create a Python program named `sol6.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./target6 $(python sol6.py)
```

target7: Return-to-libc [extra credit]

(*Difficulty: Hard*)

This target is identical to `target2`, but it is compiled with DEP enabled. Implement a return-to-libc–style attack to bypass DEP and open a root shell. You might want to use the ROPgadget tool (https://github.com/JonathanSalwan/ROPgadget) to create your gadgets.

1. There are many ways to implement a return-oriented program, but your ROP should use the `execve` syscall to run the "/bin/sh" binary: `execve("/bin/sh", 0, 0);`.

2. To help you get started, `int 0x80` is the assembly instruction for interrupting execution with a syscall, and if the EAX register contains the number 11, it will be an execve. Your job is to figure out what values you need for EBX, ECX, and EDX, and set them using ROP gadgets!

What to submit Create a Python program named `sol7.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./target7 $(python sol7.py)
```

You may find the `objdump` utility helpful.

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

target8: Heap-based exploitation [Extra credit] (*Difficulty: Hard*)

This program implements a doubly linked list on the heap. It takes three command-line arguments. Figure out a way to exploit it to open a root shell. You may need to modify the provided shellcode slightly.

What to submit Create a Python program named `sol8.py` that print lines to be used for each of the command-line arguments to the target. Your program should take a single numeric argument that determines which of the three arguments it outputs. Test your program with the command line:

```
./target8 $(python sol8.py 1) $(python sol8.py 2) $(python sol8.py 3)
```

target9: Callback shell [Extra credit] (*Difficulty: Hard*)

This target uses the same code as `target3`, but you have a different objective. Instead of opening a root shell, write your own shellcode that implements a *callback shell*. Your shellcode should open a TCP connection to `127.0.0.1` on port `31337`. Commands received over this connection should be executed at a root shell, and the output should be sent back to the remote machine.

What to submit Create a Python program named `sol9.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./target9 $(python sol9.py)
```

For the remote end of the connection, use `netcat`:

```
nc -l 31337
```

To receive credit, you must include (as an extended comment in your Python file) a fully annotated disassembly of your shellcode that explains in detail how it works.

Submission Checklist

Join a Canvas group with your partner. Upload to Canvas a gzipped tarball (`.tar.gz`) named `project1.`*`PennKey1`*`.`*`PennKey2`*`.tar.gz`. The tarball should contain only the files below:

`cookie` [Generated by `setcookie` based on your PennKeys.]
`sol0.py`
`sol1.py`

`sol2.py`
`sol3.py`
`sol4.py`
`sol5.py`
`sol6.py`
`sol7.py` [Optional extra credit.]
`sol8.py` [Optional extra credit.]
`sol9.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. Do not include `shellcode.py` with your submission. Be sure to test that your solutions work correctly in the provided VM without installing any additional packages.