Assignment 2: Application Security

This project is due on **Friday, October 28 at 6 p.m.**. You will work in **teams of two** and submit one project per team. Please find a partner as soon as possible. If you have trouble forming a team, post to Piazza's partner search forum.

The code and other answers your group submits must be entirely your own work. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you must 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 by one of the group members. Details on the submission guideline are listed at the end of the document.

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.

- 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!** Per the course ethics policy, you are required to respect the privacy and property rights of others at all times, *or else you will fail the course.*

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 at `https://www.cs.princeton.edu/~sa8/cos432/AppSec.ova`. This file is 1.4 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 password `ubuntu`.
5. Download `https://www.cs.princeton.edu/courses/archive/fall16/cos432/hw2/targets.tar.gz` from inside the VM. This file contains all of the programs you will exploit.
6. Decompress `targets.tar.gz` with `tar xf targets.tar.gz`
7. `cd targets`
8. Each group's programs will be slightly different. Personalize the programs by running: `./setcookie` *`netid`*
   Use the netid of the person who will be submitting your team's solution. Make sure the netid is correct! If you are changing your cookie, make sure to `make clean` first and then recompile!
9. `sudo make` (The password you're prompted for is `ubuntu`.)

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`.

Control Hijacking Before you begin this project, it is advised to read "Smashing the Stack for Fun and Profit" available at `https://www.cs.princeton.edu/courses/archive/fall16/cos432/hw2/stack_smashing.pdf`. It was published 20 years ago, so don't expect all the low-level details to be completely compatible with modern systems. However, it is a classic paper on the topic of buffer overflows and a great read for understanding variances of these attacks.

GDB You will make extensive use of the GDB debugger. Useful commands 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:
`https://www.cs.princeton.edu/courses/archive/fall16/cos432/hw2/gdb-refcard.pdf`

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. 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 provided programs for this project are simple, short C programs with (mostly) clear security vulnerabilities. We are going to refer to these vulnerable programs as "targets". We have provided source code and a Makefile that compiles all the targets. Your solutions must work against these targets as compiled and executed within the provided VM.

target0: Overwriting a variable on the stack

This program takes input from `stdin` and prints a message. Your job is to provide input that makes it output: "`Hi` *`netid`*`! 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. 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.

Solution: 10 bytes are allocated in the stack for the local buffer name and the next 5 bytes are allocated for buffer grade. So, for input to target, we give the netid and fill with zero until we have 10 bytes and then put "A+" to overwrite the grade buffer.

What to submit Create a Python 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
```

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

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?
   Extra Hints:
   (a) *`gdb target1`*
   (b) *`disas print_good_grade`*
   (c) *Starting address: 0x08048efe*
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?
   Extra Hints:
   (a) *`disas vulnerable`*
   (b) *lea -0xc(%ebp),%eax instruction shows that input[] is stored 12 bytes away from the base pointer(or else frame pointer). Frame pointer points to the previous frame pointer (4 bytes). After the previous frame pointer the return address(4 bytes) is kept. So input should be 20 bytes long.*
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?
   Extra Hint:
   (a) *The return address should be equal to the starting address of print_good_grade (0x08048efe)*

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

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's the current value of `%ebp`? What will be the starting address of the shellcode?
   Extra Hints:
   (a) *lea -0x6c(%ebp),%eax instruction shows that buff begins 108 (0x6c) bytes away from the base pointer.*
   (b) *`info reg $ebp` (current value of $ebp is 0xbffe9988)*
   (c) *Starting address of the shellcode will be 0xbffe9988 - 0x6c*
6. Identify the address after the call to `strcpy` and set a breakpoint there:
   `(gdb) break *addr`
   Continue the program until it reaches that breakpoint.
   `(gdb) cont`
   Extra Hint:
   (a) *In the disassembled vulnerable the instruction address after the function call is 0x08048efb*
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.
   Extra Hint:
   (a) *The return address is stored at $ebp+4. Need to replace it with the starting address of the shellcode, Start of shellcode and return address are 112 bytes apart. So, we need to fill (112-len(shellcode)) bytes with an arbitrary value.*

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

In this target, the programmer is using a safer function (`strncpy`) to copy the input string to a buffer. Therefore, the buffer overflow exploit is restricted and cannot directly overwrite the return address. However, this programmer has miscalculated the length of the buffer. Hopefully this will help you to find another way to gain contorl. Your input should cause the provided shellcode to execute and open a root shell.

*Solution: Examine target3.c . strncpy copies sizeof(buf)+8 bytes, so we can overwrite pointer p and integer a. The goal is to overwrite the return address with the starting address of shellcode, same as in target2. Given the assignment *p=a , we just need to overwrite a with the starting address of the shellcode (%ebp-0x810 found by examining the assembler code for vulnerable as in target2) and p with the address that the return address is stored at(%ebp+4).*

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

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.

Solution: The 32-bit count will determine how much memory is allocated. With alloca() the newly allocated block is placed on top of stack by adjusting the stack pointer and the return value is a pointer to the beginning of the allocated space. There no check for stack overflow (but the act of accessing illegal memory space will lead to segmentation fault). So, we can allocate as much space as we want. By allocating space equal to the whole memory(count=0xFFFFFFFF), the start of the allocated memory will be the address of buf. Thus, we can overwrite all the local variables, the previous base pointer and then the return address.
The process of finding the address of buf is:

1. *python sol4.py > tmp; gdb -args ./target4 tmp*
2. *b read_file*
3. *run*
4. *Examine target4.c . Notice that the buf pointer is the last local variable that is defined, so at the current breakpoint the stack pointer should point to this variable*
5. *Do info reg $esp to get the value of the stack pointer (0xbffe9950)*

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

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.

Solution: We need to overwrite the return address of the vulnerable function with the address of a system call. The input to the system call should be "/bin/sh" so as to open a root shell. The system call will take as argument the string that the top of the stack points at. Thus, we need to place "/bin/sh" somewhere in the stack and then put its address at the top of the stack.
Specific steps:

1. *gdb --args ./target5 $(python sol5.py)*
2. *disas vulnerable*
3. *lea -0x12(%ebp),%eax instruction shows that buff begins 18 bytes away from the base pointer(%ebp).*
4. *So in order to reach the return address from the beginning of the buf we need to write with arbitrary value (18+4) bytes. The extra 4 overwrite the previous stack pointer.*
5. *We need to find now the system call address to overwrite the return address*
6. *disas greetings*
7. *The address of the system call is 0x08048eed*
8. *After overwriting the address of the system call we need to write to the stack the address of the string "/bin/sh" (top of the stack after returning) . We will place the string in address %ebp+12. (4 bytes for the prev frame pointer, 4 for return address, 4 for its address)*

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.

target6: Variable stack position

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 0x10–0x110 bytes each time it runs. You need to construct an input that always opens a root shell despite this randomization.

Solution: The high level idea is that we are not sure where the overflow buffer starts, and so in order to avoid a segmentation fault we will fill the overflow buffer with NOPs and put the shellcode somewhere in the middle. If the return address points anywhere in the string of NOPs then the NOPs will get executed until our shellcode is reached.
Specific points:

1. *We will debug the program once and find the address of the overflow buffer (buf) for this run (let's call it buf_addr).*
2. *The address of the buf for every run will always be within the range [buf_addr-0x100,buf_addr+0x100].*
3. *The distance between the start of the buf and the %ebp is always constant though*
4. *From the assembler code of vulnerable we look for instruction lea and find that buf is 0x408 bytes away from %ebp and thus 0x40c from the return address.*
5. *The NOP instruction is one byte long and translates to 0x90 in machine code*
6. *Fill overflow buffer with NOPs and put the shellcode somewhere in the middle taking into account the offset range(0x100)*

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. **Your solution must not cause the program to print out any error messages**. Test your program with the command line:

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

Submission Checklist

Upload to this link the following files:

- `partners.txt` [One netid on each line]
- `cookie` [Generated by `setcookie` based on your netid.]
- `sol0.py`
- `sol1.py`
- `sol2.py`
- `sol3.py`
- `sol4.py`
- `sol5.py`
- `sol6.py`
- `README`

The README file should include a small explanation of your approach for every target.

Your files can make use of standard Python libraries and the provided `shellcode.py`, but they must be otherwise self-contained. Do not submit any additional files. Be sure to test that your solutions work correctly in the provided VM without installing any additional packages.