MP 2: Application Security

The first Checkpoint is due on **October 4, 2017** at **11:59 PM**, the second checkpoint is due on **October 11, 2017** at **11:59 PM**. The first checkpoint is relatively easy, and is intended to get you started. We strongly recommend that you get started early.

This is a group project; you SHOULD work in **teams of two** and if you are on a team of two, you MUST submit one project for your team. Please find a partner as soon as possible. If you have trouble forming a team, post to Piazza. Not all groups will finish all the tasks in all the MPs. The tasks in each MP are designed to be progressively harder, with the final tasks in each MP having been designed as **significant** challenges.

The code and other answers your group submits MUST be entirely your own work, and you are bound by the Student Code. 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 in the SVN directory of one (not both) of the group members, following the submission checklist given at the end of each checkpoint. Details on the filename and submission guidelines are listed at the end of this document.

"*History has taught us: never underestimate the amount of money, time, and effort someone will expend to thwart a security system.*"

– Bruce Schneier

Introduction

This project will introduce you to control-flow hijacking vulnerabilities in application software, including buffer overflows. You will be working through this MP 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 for 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!** 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*. See the "Ethics, Law, and University Policies" section on the course website.

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 macOS.
2. Get the VM file from `https://uofi.box.com/v/cs461vm`. This file is 1.3 GB, so we recommend downloading it from the campus network.
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. We have generated blank submission files for you in your SVN repo. Check out the files with this command:
   `svn checkout https://subversion.ews.illinois.edu/svn/fa17-cs461/NETID/mp2`

6. Download `https://subversion.ews.illinois.edu/svn/fa17-cs461/_shared/mp2/mp2.tar.gz` from inside the VM. This file contains all of the programs for both checkpoints.

7. Put `mp2.tar.gz` inside your mp2 folder that you checked out from SVN.

8. Decompress `mp2.tar.gz` in your mp2 folder with `tar xf mp2.tar.gz`

9. Each person's solutions will be slightly different. You MUST personalize the programs by running:
   `./setcookie` *`netid`*
   Use the NetID of the repository in which you will be submitting your team's solution. MAKE SURE the NetID is correct! IF YOU WANT TO CHANGE YOUR COOKIE, make sure to run `make clean` first and then recompile!

10. Run `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`.

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://csapp.cs.cmu.edu/2e/docs/gdbnotes-ia32.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`

2.1 Checkpoint 1 (20 points)

The practice programs for this project are designed to let you get familiar with GDB, the stack frame for C functions, x86 assembly, and Linux system calls so you have the tools to tackle checkpoint 2. We have provided source code and a Makefile that compiles all the programs for both checkpoint 1 and checkpoint 2. Your solutions MUST work when compiled and executed within the provided VM.

2.1.1 GDB Practice (4 points)

This program really doesn't do anything on its own, but it allows you to practice using GDB and look at what the program is doing at a lower level. You have two jobs here. Your first job is to figure out where the function `practice` returns to. Your second job is to find out the value in register `eax` right before the `practice` function returns. Here's one approach you might take:

1. Start the debugger (`gdb`) and set a breakpoint at function `practice`:
   `(gdb) b practice`, then run the program: `(gdb) r`.
2. Think about where the return address is relative to register `ebp` (the base pointer).
3. Examine (`x` in gdb) that memory location: `(gdb) x 0xaddress` or `(gdb) x $ebp+#` and put the contents of the address in `2.1.1_addr.txt`.
   Remember that you can use `(gdb) info reg` to look at the values of registers at a breakpoint!
4. Disassemble `practice` with `(gdb) disas practice` then set a breakpoint at the address of the `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 the breakpoint: `(gdb) c`.
5. With `(gdb) info reg`, you can view the contents of `eax` and put the value in `2.1.1_eax.txt`.

What to submit Submit the return address of the function `practice` in `2.1.1_addr.txt` and the value of `eax` right before `practice` returns in `2.1.1_eax.txt`. You MUST submit both in hexadecimal representation (the `0x` prefix is optional).

2.1.2 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 arguments passed to a C function?

2. In what order should the arguments be pushed onto the stack?

Tips:

1. Use `push $0x12341234` to push an arbitrary hex value onto the stack.
2. Use `call function_name` to call functions.

What to submit Submit your x86 assembly code in `2.1.2.S`. Make sure the entire program exits properly with your assembly code!

2.1.3 Assembly Practice with Pointers (4 points)

Just like `2.1.2`, your goal is to call the function `practice` with the correct arguments so that the function prints out "`Good job!`". Note that the parameters are slightly different from `2.1.2`. Hint: Think about what would be on top of the stack if you do:

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

What to submit Submit your x86 assembly code in `2.1.3.S`. Make sure the entire program exits properly with your assembly code!

2.1.4 Assembly Practice with Pointers and Strings (4 points)

Just like `2.1.2` and `2.1.3`, 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 in `2.1.2` and `2.1.3`. Tips:

1. The byte order on x86 is little endian.
2. Characters are read on the stack from top to bottom (low address to high address).
3. What character/value indicates the end of a string?

What to submit Submit your x86 assembly code in `2.1.4.S`. Make sure the entire program exits properly with your assembly code!

2.1.5 Introduction to Linux System Calls (4 points)

Your goal for this exercise is to invoke a system call using `int 0x80` to open a shell. Tips:

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

2. The function signature of `sys_execve` in C:
   `int execve(const char *filename, char *const argv[], char *const envp[]);`
3. Instead of passing the arguments on 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 (in order) `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp`.
6. To start a shell, the first argument is the filename (a string) which should be something like `/bin/sh`.
7. Reading the Linux man pages may help.
8. Some arguments may need to be terminated with a null character/pointer.

What to submit Submit your x86 assembly code in `2.1.5.S`.

Checkpoint 1: Submission Checklist

The following blank files for checkpoint 1 have been created in your SVN repository in the `mp1` directory. Put your cookie and solutions in the corresponding files, then commit them to SVN.

- `partners.txt` [One NetID on each line]
- `cookie` [Generated by `setcookie` based on your NetID]
- `2.1.1_addr.txt`
- `2.1.1_eax.txt`
- `2.1.2.S`
- `2.1.3.S`
- `2.1.4.S`
- `2.1.5.S`

Do not add any unnecessary files to your repository. Be sure to test that your solutions work correctly in the provided VM without installing any additional packages.

2.2 Checkpoint 2 (100 points)

Again, we have provided source code and a Makefile that compiles all the programs for both checkpoint 1 and checkpoint 2. We are going to refer to these vulnerable programs as "targets" for the rest of this MP. Your solutions MUST work against these targets as compiled and executed in the provided VM.

2.2.1 Overwriting a Variable on the Stack (8 points) (*Difficulty: Easy*)

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 `2.2.1.c`. Where is the buffer overflow?
2. Start the debugger (`gdb`) 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 `./2.2.1` on the command line with different inputs.

What to submit Write a Python program in `2.2.1.py` that prints a line to be passed as input to the target. Test your program with the command:

```
python 2.2.1.py | ./2.2.1
```

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`.

2.2.2 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 `2.2.2.c`. Where is the buffer overflow?
2. Disassemble `print_good_grade`. At what address does it start?
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`? What length of input will 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 address `%ebp` using: `(gdb) x/2wx $ebp`

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

What to submit Write a Python program in `2.2.2.py` that prints a line to be passed as input to the target. Test your program with the command:

```
python 2.2.2.py | ./2.2.2
```

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

```
python 2.2.2.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)
```

2.2.3 Redirecting Control to Shellcode (8 points) (*Difficulty: Easy*)

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

1. Examine `2.2.3.c`. Where is the buffer overflow?

2. Create a Python program in `2.2.3.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 ./2.2.3 $(python 2.2.3.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 the breakpoint.
   ```
   (gdb) cont
   ```

7. Examine the bytes of memory where you think the shellcode is to confirm your calculations: `(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 Write a Python program in `2.2.3.py` that prints a line to be used as the command line argument to the target. Test your program with the command:

```
./2.2.3 $(python 2.2.3.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 ./2.2.3 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 ./2.2.3` in order for the core dump to be created.

2.2.4 Overwriting the Return Address Indirectly (9 points) (*Difficulty: Medium*)

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

What to submit Write a Python program in `2.2.4.py` that prints a line to be used as the command line argument to the target. Test your program with the command:

```
./2.2.4 $(python 2.2.4.py)
```

2.2.5 Beyond Strings (9 points) (*Difficulty: Medium*)

This target takes as its command line argument the name of a data file to 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 open a root shell.

What to submit Write a Python program in `2.2.5.py` that outputs the contents of the data file to be read by the target. Test your program with the command:

```
python 2.2.5.py > tmp; ./2.2.5 tmp
```

2.2.6 Bypassing DEP (9 points) (*Difficulty: Medium*)

This program resembles `2.2.3`, 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 Write a Python program in `2.2.6.py` that prints a line to be used as the command line argument to the target. Test your program with the command:

```
./2.2.6 $(python 2.2.6.py)
```

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

2.2.7 Variable Stack Position (9 points) (*Difficulty: Medium*)

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 `2.2.3`, 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.

What to submit Write a Python program in `2.2.7.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:

```
./2.2.7 $(python 2.2.7.py)
```

2.2.8 Linked List Exploitation (10 points) (*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 Write a Python program in `2.2.8.py` that prints a line to be used for each of the command line arguments to the target. Test your program with the command:

```
./2.2.8 $(python 2.2.8.py)
```

2.2.9 Return-Oriented Programming (10 points) (*Difficulty: Hard*)

This target uses the same code as `2.2.3`, but it was compiled with DEP enabled. Your job is to construct a ROP attack to open a root shell. Tips:

1. You can use `objdump` to search for useful gadgets:
   `objdump -d ./2.2.9 > 2.2.9.txt`

2. Reading Havoc's paper may help: `https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf`

What to submit Write a Python program in `2.2.9.py` that prints a line to be used as the command line argument to the target. Test your program with the command:
`./2.2.9 $(python 2.2.9.py)`

2.2.10 Callback Shell (10 points)

(*Difficulty: Hard*)

This target uses the same code as `2.2.4`, but you have a different objective. Instead of opening a root shell, write your own shellcode to implement a *callback shell*. Your shellcode should open a TCP connection to `127.0.0.1` on port `31337`. **ONLY this exact address and port combination will be accepted as correct.** Commands received over this connection should be executed at a root shell, and the output should be sent back to the remote machine. Tips:

1. Make sure you understand what this is doing:

```
int sockfd;
struct sockaddr_in addr;

addr.sin_family = AF_INET;
addr.sin_addr.s_addr =
     inet_addr(SERV_HOST_ADDR);
addr.sin_port = htons(SERV_TCP_PORT);

sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, (struct sockaddr *) &addr,
     sizeof(serv_addr));
do_stuff(stdin, sockfd);
```

2. Network byte order for `sin_addr` and `sin_port` are big endian.
3. You can write your shellcode in x86 assembly, then use objdump to translate the shellcode to hex.

What to submit Write a Python program in `2.2.10.py` that prints a line to be used as the command line argument to the target. Test your program with the command:
`./2.2.10 $(python 2.2.10.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 explaining how it works in detail.

2.2.11 Format String Attack (10 points) (*Difficulty: Medium*)

Did you know that `printf` is actually vulnerable to a type of attack called a format string attack? Your job is to exploit this and open a root shell. You should think about what the format specifier `%n` does.

What to submit Write a Python program in `2.2.11.py` that prints a line to be used as the command line argument to the target. Test your program with the command:

```
./2.2.11 $(python 2.2.11.py)
```

Checkpoint 2: Submission Checklist

The following blank files has been created in your SVN repository in the `mp2` directory. Put your cookie and solutions in the corresponding files, then commit them to SVN.

- `partners.txt` [One NetID on each line]
- `cookie` [Generated by `setcookie` based on your NetID]
- `2.2.1.py`
- `2.2.2.py`
- `2.2.3.py`
- `2.2.4.py`
- `2.2.5.py`
- `2.2.6.py`
- `2.2.7.py`
- `2.2.8.py`
- `2.2.9.py`
- `2.2.10.py`
- `2.2.11.py`

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