Project 1: Application Security

This project is split into two parts, with the first checkpoint due on **Monday, Febuary 6, 2023, 6:00 PM**. and the second checkpoint due on **Friday, Febuary 10, 2023, 6:00 PM**. All MPs will be graded out of 120 points. The first checkpoint is worth 20 points and the second checkpoint is worth 100. We strongly recommend that you get started early, as we expect Checkpoint 2 to take significantly more time to complete than Checkpoint 1.

This is a group project; you SHOULD work in **teams of two**, and if you are in teams of two, you MUST submit one project per team. Please find a partner as soon as possible. If you have trouble forming a team, post to Canvas's partner search forum.

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 group member's Git repos, following the submission checklist given at the end of each checkpoint. Details on the filenames and submission guidelines are listed at the end of the 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, such as buffer overflows, in application software. 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 by 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 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 will use this precise configuration to grade your submissions, so you MUST NOT use your own VM or host. Additionally, you MUST use the default version of Python on the VM to write your exploits (Python 3.6.9).

1. Set up the CS461 VM. We have created a Canvas post which walks you through this process. Search Canvas for "Virtual Machine Setup and Troubleshooting".
2. **Do not, at any point, run** `sudo apt(-get) upgrade` on this VM or install updates via the graphical update manager. Doing so may unintentionally re-enable some security features on the VM, preventing your exploits from working properly.
3. Clone your git repository onto your VM:
   `git clone https://github.com/illinois-cs-461/sp23_cs461_<netid>.git`. You may need to setup your git credentials for the VM. For https access, you may need to setup a personal token, and authorize the key for single sign-on within illinois-cs-461. If you

choose to setup a ssh key, please remember to authorize the token for single sign-on within illinois-cs-461. Here are references you may find useful:
`https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token`
`https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account`
`https://docs.github.com/en/enterprise-cloud@latest/authentication/authenticating-wi authorizing-an-ssh-key-for-use-with-saml-single-sign-on`

4. Once your repo is set up, course staff will generate blank submission files in a new branch of your repo called *`AppSec`*. Merge the *`AppSec`* branch into the *`master`* branch by running the following commands:
```
git pull
git merge origin/AppSec
git push origin master
```

5. From inside the VM, visit the Assessments page of the course website and download `AppSec.tar.gz`. This file contains all of the programs for both checkpoints.

6. Put `AppSec.tar.gz` inside your AppSec folder which has been created within your git repo.

7. Decompress `AppSec.tar.gz` in your AppSec folder with the following command:
```
tar -zxvf AppSec.tar.gz
```

8. Each person's solutions will be slightly different. You MUST personalize the programs by running:
`./setcookie` *`netid`*

   IMPORTANT—Partners and Grading: Each repository will be graded using the cookie corresponding to that repository's netid. Although we will autograde the repository of each partner, the higher of the two grades will be assigned to both partners. We recommend selecting one of your two repositories and working only on solutions for that one. In this case, both partners should set their cookie based on the netid of that single repository. This will ensure that solutions developed by one partner work for the other partner as well.

   Example: Assume that Paul (netid: *pmurley2*) and Joshua (netid: *joshuar3*) have decided to be partners. They decide they will use Joshua's repository as their primary one for submission. Before Joshua compiles, he sets his cookie with: `./setcookie joshuar3`. Although Paul may use his repository (*pmurley2*) for storing his own work, **he also uses**: `./setcookie joshuar3` before he compiles, so that the solutions he develops will be correct for the *joshuar3* repository. Both repositories are graded. The *pmurley2* repository will receive a very low score, since the solutions were not developed with the *pmurley2* cookie. However, the *joshuar3* repository receives a 115/120. The autograder also reads the `partners.txt`

file in Joshua's repository and determines that Joshua and Paul are partners. Both Paul and Joshua receive 115/120 as their score for the MP.

9. Place the netids of **you and your partner** in `partners.txt` (one per line).

10. Once you have understood the above and properly set your cookie, run `sudo make` to build the programs. If you ever need to change your cookie, you will need to `make clean`, `./setcookie` *`netid`*, and `sudo make` again. Note that changing your cookie will likely break your solutions.

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` and `objdump`.

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! Here are a couple of possibly helpful resources on GDB:
`http://csapp.cs.cmu.edu/2e/docs/gdbnotes-x86-64.pdf`
`http://users.ece.utexas.edu/~adnan/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 32-bit architecture, not x86_64. Also note that while this project is done on the x86 architecture, an Intel design, we use exclusively AT&T syntax for assembly code, which is the default for GDB. Here are a few more references you may find useful:
`https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s3`
`http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html`
Helpful link for linux system calls:
`https://faculty.nps.edu/cseagle/assembly/sys_call.html`

Python To construct exploits for Checkpoint 2 problems, you will use Python 3.6.9, installed as the default version of `python3` on the VM. You may find the following example useful:

```
1  import sys
2  from struct import pack
3  from shellcode import shellcode
4
5  sys.stdout.buffer.write(b"A" * 90 + shellcode + pack("<I", 0xDEADBEEF))
6
```

We recommend using this `sys.stdout.buffer.write()` function to output your payloads. You can write literal binary strings by putting a "b" character directly before the string, and you can multiply those strings by integers to duplicate them. The `struct.pack()` function provides a convenient way to pack hex addresses into little-endian formatting and directly concatenate them with other binary strings.

1.1 Checkpoint 1 (20 points)

The Checkpoint 1 programs for this project are designed to let you get familiar with GDB, stack-frames 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 as compiled and executed within the provided VM.

1.1.1 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 is to determine what value is in register `eax` **right before** the function `practice` returns.

Here's one approach you might take:

1. Start the debugger (`gdb 1.1.1`), 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 `1.1.1_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 `1.1.1_eax.txt`.

What to submit Submit the return address of the function `practice` in `1.1.1_addr.txt` and the value of `eax` right before `practice` returns in `1.1.1_eax.txt`. You MUST submit both in hexadecimal representation and in lower case (0x prefix is optional).

1.1.2 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 arbitrary hex value onto the stack.
2. Use `call function_name` to call functions.

What to submit Submit your x86 assembly code in `1.1.2.S`. Make sure the entire program exits properly with your assembly code!

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

Just like `1.1.2`, 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 `1.1.2`.

Hint: Think about what would be on top of the stack if you run the following instructions:

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

What to submit Submit your x86 assembly code in `1.1.3.S`. Make sure the entire program exits properly with your assembly code!

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

Just like `1.1.2` and `1.1.3`, 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 `1.1.2` and `1.1.3`.

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. You can check this for help. `https://upload.wikimedia.org/wikipedia/commons/thumb/e/ed/Little-Endian.svg/2000px-Little-Endian.svg.png`
4. What character/value indicates end of string?

What to submit Submit your x86 assembly code in `1.1.4.S`. Make sure the entire program exits properly with your assembly code!

1.1.5 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.

What to submit Submit your x86 assembly code in `1.1.5.S`.

Checkpoint 1: Submission Checklist

The following blank files for Checkpoint 1 have been created in your Git repository under the directory AppSec. Ensure you are working on the *master* branch of your repository by typing: `git branch`. Put your cookie and solutions inside the corresponding files. Commit, push, and ensure your solutions are properly uploaded by viewing your repository through the web interface (`https://github.com/illinois-cs-461/sp23_cs461_<netid>.git`).

- `partners.txt` [One netid on each line]
- `cookie` [Generated by `setcookie` based on your netid.]
- `1.1.1_addr.txt`
- `1.1.1_eax.txt`
- `1.1.2.S`
- `1.1.3.S`

- `1.1.4.S`
- `1.1.5.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.

1.2 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 the MP. Your solutions MUST work against these targets as compiled and executed within the provided VM. Run your solutions as shown in the handout. You should NOT have to put quotes around your command line argument(s) (e.g. `./1.2.6 "$(python3 1.2.6.py)"`). If your solution only works when run like this, then your solution does not work.

1.2.1 Overwriting a variable on the stack (10 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 `1.2.1.c`. Where is the buffer overflow?
2. Start the debugger (`gdb 1.2.1`) 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 `./1.2.1` on the command line with different inputs.

What to submit Create a Python program named `1.2.1.py` that prints a line to be passed as input to the target. Test your program with the command line:

```
python3 1.2.1.py | ./1.2.1
```

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

1.2.2 Overwriting the return address (10 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 `1.2.2.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 `1.2.2.py` that prints a line to be passed as input to the target. Test your program with the command line:

```
python3 1.2.2.py | ./1.2.2
```

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

```
python3 1.2.2.py | hd
```

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

```
from struct import pack
import sys
sys.stdout.buffer.write(pack("<I", 0xDEADBEEF))
```

1.2.3 Redirecting control to shellcode (10 points) (*Difficulty: Easy*)

Your goal for this and all remaining targets is to cause them to launch a shell. 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 `1.2.3.c`. Where is the buffer overflow?

2. Create a Python program named `1.2.3.py` that outputs the provided shellcode:
   ```
   import sys
   from shellcode import shellcode
   sys.stdout.buffer.write(shellcode)
   ```

3. Set up the target in GDB using the output of your program as its argument:
   `gdb --args ./1.2.3 $(python3 1.2.3.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 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 `1.2.3.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./1.2.3 $(python3 1.2.3.py)
```

If you are successful, you will see a shell prompt (`$`).

1.2.4 Overwriting the return address indirectly (10 points) (*Difficulty: Medium*)

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 control. Your input should cause the provided shellcode to execute and open a shell.

What to submit Create a Python program named `1.2.4.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./1.2.4 $(python3 1.2.4.py)
```

1.2.5 Beyond strings (10 points) (*Difficulty: Medium*)

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 shell.

What to submit Create a Python program named `1.2.5.py` that outputs the contents of a data file to be read by the target. Test your program with the command line:

```
python3 1.2.5.py > tmp; ./1.2.5 tmp
```

1.2.6 Bypassing DEP (10 points) (*Difficulty: Medium*)

This program resembles `1.2.3`, 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 shell.

What to submit Create a Python program named `1.2.6.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./1.2.6 $(python3 1.2.6.py)
```

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

1.2.7 Variable stack position (10 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 `1.2.3`, 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 shell despite this randomization.

What to submit Create a Python program named `1.2.7.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 line:

```
./1.2.7 $(python3 1.2.7.py)
```

1.2.8 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 shell. You may need to augment the provided shellcode slightly in your 1.2.8.py, but you should not alter `shellcode.py`.

IMPORTANT NOTE: Because heap address may vary across virtual machines, you must determine and submit the addresses (on the heap) of each of your three nodes (*a*, *b*, and *c*) on the lines provided in your Python file. This will assist the autograder in accounting for differences between your heap addresses and those on the autograder.

What to submit Create a Python program named `1.2.8.py` that print lines to be used for each of the command-line arguments to the target. Test your program with the command line:

```
./1.2.8 $(python3 1.2.8.py)
```

1.2.9 Format String Attack (10 points) (*Difficulty: Hard*)

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

Tips:

1. You can work on the proto-answer as: sys.stdout.buffer.write(shellcode + padding + ADDR1 + ADDR2 + b"%_____x%__$hn%_____x%__$hn").
2. Figure out what goes in each field. How much padding do you need? What are the two addresses? What number goes in each blank of the proto-answer?
3. You may also want to understand "Direct Parameter Access" at `https://cs155.stanford.edu/papers/formatstring-1.2.pdf`

What to submit Create a Python program named `1.2.9.py` that prints a line to be used as the command-line argument to the target. Test your program with the command line:

```
./1.2.9 $(python3 1.2.9.py)
```

1.2.10 Returned-oriented Programming (10 points) (*Difficulty: Hard*)

This target uses the same code as `1.2.3`, but it is compiled with DEP enabled. Your job is to contruct a ROP attack to open a shell.

Tips:

1. You can use objdump to search for useful gadgets:
   `objdump -d ./1.2.10 > 1.2.10.txt`
2. Reading Hovav Shacham's paper may help: `https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf`

Checkpoint 2: Submission Checklist

The following files has been created in your git repository under the directory AppSec. Put your cookie and solution inside the corresponding file and then commit and push each solution when complete.

- `partners.txt` [One netid on each line]
- `cookie` [Generated by `setcookie` based on your netid.]
- `1.2.1.py`
- `1.2.2.py`
- `1.2.3.py`
- `1.2.4.py`
- `1.2.5.py`
- `1.2.6.py`
- `1.2.7.py`
- `1.2.8.py`
- `1.2.9.py`
- `1.2.10.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.