Project 4: Application Security

This project is due on **November 15, 2018** at **6 p.m.** and counts for 8% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 5 hours until received. Late work will not be accepted after 20.5 hours past the deadline. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

This is a group project; 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, and you are bound by the Honor Code. 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 below. Please coordinate carefully with your partner to make sure at least one of you submits on time.

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
- Learn to use popular reverse engineering tools to dissect a binary

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 a Kali Linux VM in which you should develop and test your attacks. We've also slightly tweaked the configuration to disable 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. You must download the targets into a folder owned by the VM, running targets in a shared folder will cause incorrect behavior.

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://eecs388.org/!/dist/388-kali-p4.ova. This file is 2 GB, so we recommend downloading it from campus.
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 `eecs388` and `eecs388`.
5. Download https://eecs388.org/project4/targets.tar.gz from inside the VM. This file contains the target programs you will exploit.
6. `tar -xzvf targets.tar.gz`
7. `cd targets/`
8. Each group's targets will be slightly different. Personalize the targets by running:
   `./setcookie` *`uniqname1 uniqname2`*
   Make sure both uniqnames are correct!
9. `sudo make` (The password you're prompted for is `eecs388`. *You must run this command as sudo*)

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 `gdb` or `idapro`.

Control Hijacking Before you begin this project, review the lecture slides from the control-hijacking lecture and attend discussion for additional details. Read "Smashing the Stack for Fun and Profit," available at https://eecs388.org/*/stack_smashing.pdf.

Ida Pro You will become familiar with the basics of using Ida Pro, the reverse engineering tool of choice for antivirus companies such as Symantec. You can download the freeware version at https://www.hex-rays.com/products/ida/support/download.shtml. It will not run within the 32-bit VM, which is fine because the free version does not support debugging anyways. Instead, you will be using IDA for performing static analysis from your own machine.

Some useful tutorials/references:
https://resources.infosecinstitute.com/basics-of-ida-pro-2/#gref
https://jlospinoso.github.io/developing/software/software%20engineering/reverse%20engineering/assembly/2015/03/06/reversing-with-ida.html

Useful hotkeys/tips:
"g": go to a given hex address
"[space bar]": switch to and from graph mode
"[alt] t": search deadlisting text for a string
"[escape]": go back
"[shift][enter]": forward
";": write a comment
"x": find all cross references
double click a label: jump to that location
single click a label: highlight all occurences

GDB You will make use of the GDB debugger for dynamic analysis within the VM, which you should recall from EECS 280. Useful commands that you may not know are "`disassemble`", "`info reg`", "`x`", and "`stepi`". See the GDB help for details, and don't be afraid to experiment! This quick reference may also be useful: https://web.eecs.umich.edu/~sugih/pointers/Gdb-reference-card.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.

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)*

This program takes input from `stdin` and prints a message. Your job is to provide input that causes the program to output: "`Hi` *`uniqname`*`!  Your grade is A+.`" (You can use either group member's uniqname.) 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. Open up the binary in IDA and examine the `main`() function. 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 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 *(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 `target1.c`. Where is the buffer overflow?
2. Examine the function `print_good_grade`. What is its starting address?
3. Using GDB from within the VM, 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 would an input have to be to 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*)

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 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. Examine `vulnerable` in IDA. Where does `buf` begin relative to `%ebp`? What is the offset from the start of the shellcode to the saved return address?

4. Set up the target in GDB using the output of your program as its argument:

   ```
   gdb --args ./target2 $(python sol2.py)
   ```

5. Set a breakpoint in `vulnerable` and start the target.

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*)

In this target, the buffer overflow is restricted and cannot directly overwrite the return address. You'll 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*)

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*)

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.

Warning: do not try to create a solution that depends on you manually setting environment variables. You cannot assume that the autograder will run your solution with the same environment variables that you have set.

target6: Variable stack position (*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 starting location 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)
```

A word of caution If you see any output from the program before a root shell is opened, you have not done target6 of the project correctly and your solution will not be accepted by the autograder.

target7: Return-oriented programming (*Difficulty: Hard*)

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. It may be helpful to use a tool such as ROPgadget(https://github.com/JonathanSalwan/ROPgadget).

1. Though there are a number of ways you could implement a return-oriented program, your ROP should use the execve system-call to run the "/bin/sh" binary. This is equivalent to:

   execve("/bin/sh", 0, 0);

2. For an extra push in the right direction, 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. Now all you need to figure out is 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 prints 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, implement your own shellcode to implement 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 in a 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 on your shellcode that explains in detail how it works.

Fuzz Testing

Manually reviewing source code for vulnerabilities can be laborious and time consuming, and outsiders typically cannot do it at all for closed-source software. For these reasons, both attackers and defenders often use an automated form of vulnerability discovery called “fuzz testing” or “fuzzing” that attempts to find edge-cases that the application developers failed to account for. Unlike analysis that makes use of source code (“white-box testing”), fuzzing assumes only the ability to execute the software with chosen inputs (“black-box testing”).

In fuzzing, the analyst creates a program (a “fuzzer”) that emulates a user and rapidly provides many different automatically generated inputs to the target application while monitoring for anomalous behavior (e.g., crashes or corrupted return data). When an input consistently causes anomalous behavior, the fuzzer stores it so that the analyst can investigate the problem. The anomalous behavior may be a sign that there is an exploitable vulnerability in the code path that the input exercises.

It’s not usually feasible to test with every possible input, but a clever input generation algorithm can increase the odds that the fuzzer will trigger a bug. For instance, many fuzzers start with a set of valid inputs and then corrupt them by making randomized changes, additions, or deletions.

Setup A recently founded start-up named ZCorp has hit it big by providing data analysis expertise to customers. Customers provide data to analyze in JSON format (https://www.json.org), and ZCorp charges them based on the number of JSON string values in each customer's data. In order to generate bills, they run all customer input through a JSON parser that extracts each JSON string value, along with its index.

ZCorp outsourced development of this JSON parser to an acquaintance of one of the founders. Unfortunately for ZCorp, this developer never took EECS 388, so the parser is probably highly vulnerable to exploitation. If you can find an input that causes a SEGFAULT in the parser, ZCorp can refuse to pay the inept developer until the problem has been fixed.

Goal Your goal for this portion of the project is to create a fuzzer that is capable of automatically finding an input that causes a SEGFAULT in the provided JSON parser. You can download the `jsonParser` binary from https://eecs388.org/project4/parser.tar.gz. It reads JSON data from `stdin` and writes to `stdout` and `stderr`. While the developer did not provide ZCorp with the source code for the parser, they did provide a script, `jsonParserTests.py`, that checks a set of test cases.

You are **not** required to create an exploit or understand the cause of the SEGFAULT.
You should **not** attempt to reverse-engineer the target, as this will likely be a waste of time.
Your grade will be based upon the success of the fuzzer in causing a segfault **and** whether it tests a comprehensive set of JSON inputs.

What to submit Create a Python program named `fuzzer.py` that generates inputs, invokes `jsonParser` on them, and then determines whether there was a SEGFAULT. Your program should do this repeatedly for different inputs until a SEGFAULT occurs. When this happens, it should print the input data that triggered the fault to `stdout` as a base64-encoded string and exit. You can confirm that the input causes a SEGFAULT via the command:

```
echo "base64 encoded data" | base64 -d | ./jsonParser
```

When you find an input that consistently causes a SEGFAULT, place the base64-encoded string in a text file named `fuzzInput.txt` and submit it along with your program.

Submission Checklist

Upload to Canvas a gzipped tar file named `project4.`*`uniqname1`*`.`*`uniqname2`*`.tar.gz` that contains only the files listed below. **These will be autograded, so make sure you have the proper filenames, formats, and behaviors**. Failure to work with the autograder—for any reason—will result in a 5% deduction from the maximum possible points. You can generate the tarball at the shell using this command:

```
tar -zcf project4.uniqname1.uniqname2.tar.gz cookie sol[0123456789].py \
              fuzzer.py fuzzInput.txt
```

The tarball should contain only the files below:

`cookie` [Generated by `setcookie` based on your uniqnames.]
`sol0.py`
`sol1.py`
`sol2.py`
`sol3.py`
`sol4.py`
`sol5.py`
`sol6.py`
`sol7.py`
`sol8.py` [Optional extra credit.]
`sol9.py` [Optional extra credit.]
`fuzzer.py`
`fuzzInput.txt`

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.