Project 4: Application Security

This project is due on **April 1, 2020** at **6 p.m.** and counts for 13% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 4 hours until received. Late work will not normally be accepted after the start of the next lab (of any section) following the day of the deadline, since we will be reviewing solutions at that time. If you have a medical or personal situation that may necessitate an extension, please email eecs388-staff@umich.edu

The code and other answers you submit must be entirely your own work, and you are bound by the Honor Code. You may discuss 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. You may consult published references, provided that you appropriately cite them (e.g., with program comments). Visit the course website for the full collaboration policy.

Solutions must be submitted electronically via your GitHub Classroom repo. We encourage you to verify the submission checklist at the end of this 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.

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
- Understand the mechanics of buffer overflow exploitation

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 details of the target system, so we are providing a Kali Linux VM in which you should develop and test your attacks. We've configured the VM to disable certain security features that would complicate your work.

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://files.eecs388.org/388-kali-2020.ova. Note: This file is 3 GB, but you can reuse it for Project 5 if you'd like.
3. Launch VirtualBox and select File ▷ Import Appliance to add the VM.
4. Start the VM. The username and password required to login are `eecs388` and `eecs388`.
5. ***Do not* update the software in the VM.** Updates to the compiler or C libraries might change your solutions. We will grade using exactly the versions that are pre-installed in the image.
6. Check out your starter code from GitHub inside the VM. You must do this in a folder in the native Linux filesystem. It won't work correctly if you use a shared folder located in the host OS.
7. Download https://files.eecs388.org/targets.tar.gz from inside the VM. This file contains the target programs you will exploit.
8. Extract the targets inside the starter code folder.
   `tar -xzvf targets.tar.gz`
9. Edit `./test.sh` and set your uniqname. Each student's targets will be slightly different, so make sure your uniqname is correct!
10. Run `./test.sh` to build the targets and test the (currently empty) solutions. (The password you're prompted for is `eecs388`.) The test script will report an error for each of the targets.

Resources and Guidelines

No attack tools allowed! Except where specifically noted, 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`.

Control hijacking tutorials 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://files.eecs388.org/stack_smashing.pdf.

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 our project targets use the 32-bit x86 ISA. The stack is organized differently in x86 and x64. If you are reading any online documentation, ensure that it is based on the x86 architecture, not x64.

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+.`" 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. Disassemble `_main`. What is its starting address?
3. Set a breakpoint at the beginning of `_main` and run the program.
4. Using GDB from within the VM, set a breakpoint at the beginning of `_main` and run the program.
   ```
   (gdb) break _main
   (gdb) run
   ```
5. Draw a picture of the stack. How are `name[]` and `grade[]` stored relative to each other?
6. 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 3 program named `sol0.py` that prints a line to be passed as input to the target. Test your program with the command line:

```
python3 sol0.py | ./target0
```

Hint: In Python 3, you should work with bytes rather than Unicode strings. To construct a byte literal, use this syntax: `b"\x`*`nn`*`"`, where *nn* is a 2-digit hex value. To repeat a byte *n* times, you can do: `b"\x`*`nn`*`"*n`. To output a sequence of bytes, use:

```
import sys
sys.stdout.buffer.write(b"\x61\x62\x63")
```

Don't use `print()`, because it automatically encodes whatever is being printed with default encoding of the console. We don't want our payload to be encoded, so we use `sys.stdout.buffer.write()`.

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 3 program named `sol1.py` that prints a line to be passed as input to the target. Test your program with the command line:

```
python3 sol1.py | ./target1
```

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

```
python3 sol1.py | hd
```

Remember that x86 is little endian. Use Python's `to_bytes` method to output 32-bit little-endian values like so:

```
import sys
sys.stdout.buffer.write(0xDEADBEEF.to_bytes(4, "little"))
```

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 the following 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 3 program named `sol2.py` that outputs the provided shellcode:
   ```
   from shellcode import shellcode
   import sys
   sys.stdout.buffer.write(shellcode)
   ```

3. Disassemble `vulnerable`. 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 "$(python3 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 *0x08049b77
   ```

   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 0xaddress
   ```

8. Disassemble the shellcode: `(gdb) disas/r 0xaddress,+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 3 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 "$(python3 sol2.py)"
```

If you are successful, you will see a root shell prompt (`#`). Running `whoami` will output "`root`". Running `exit` will return to your normal shell.

If your program segfaults, you can examine the state at the time of the crash using GDB with the core dump: `gdb ./target2 core`. To enable creating core dumps, run `ulimit -c unlimited`. 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 3 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 "$(python3 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 (all little endian). Create a data file that causes the provided shellcode to execute and opens a root shell.

Hint: First figure out how an attacker can cause a buffer overflow in this program.

What to submit Create a Python 3 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:

```
python3 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 3 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 "$(python3 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 3 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 "$(python3 sol6.py)"
```

Warning: If you see any output before the root shell is opened, you have not done this target 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 will be helpful to use a tool such as ROPgadget (https://github.com/JonathanSalwan/ROPgadget); this is an exception to the "no attack tools" policy. The `ROPgadget` command is already installed on the provided VM.

1. Though there are a number of ways you could implement a ROP exploit, for this target you should use the `setuid` syscall to become root, followed by the `execve` syscall to run the `/bin/sh` binary. This is equivalent to:

   ```
   setuid(0);
   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. If the EAX register contains the number 23, the syscall will be `setuid`; if it contains 11, the syscall will be `execve`. You need to figure out what values you need for EBX, ECX, and EDX, and set them using ROP gadgets!

3. We recommend that you start by getting the `execve` call to work on its own, without `setuid`. When you do this correctly, it will open a shell, but you won't be root. Then modify your solution to make it call `setuid` first, and you'll get a root shell.

What to submit Create a Python 3 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 "$(python3 sol7.py)"
```

Hint: You may find the `objdump` utility helpful.

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

Submission Details

1. Create a repo using this GitHub Classroom link.
2. Check that your repo contains the starter files. If you have any issues with GitHub Classroom, email `eecs388-staff@umich.edu` for assistance.
3. At the deadline, your repository will be cloned automatically for grading.

   If you need to submit late, you must report the hash of the commit you want graded using the late submission form. The late penalty will apply to the timestamp of the form submission, not the timestamp of the commit.

Submission Checklist

The sample code includes a script called `test.sh` that can help you check your solutions.

Make sure that you have named each file as indicated below, as they will be autograded. Additionally, make sure that all of your files are in the **top** directory of your GitHub repository and not within any subdirectories.

At the deadline, the following files should be pushed to your GitHub Classroom project repository:

- ☐ `cookie` Generated by `setcookie` based on your uniqname.
- ☐ `sol0.py`
- ☐ `sol1.py`
- ☐ `sol2.py`
- ☐ `sol3.py`
- ☐ `sol4.py`
- ☐ `sol5.py`
- ☐ `sol6.py`
- ☐ `sol7.py`

Your files can make use of standard Python 3 libraries and the provided `shellcode.py`, but they must be otherwise self-contained. Do not modify or include the targets, `Makefile`, `helper.c`, `shellcode.py`, etc. Be sure to test that your solutions work correctly in an unmodified copy of the provided VM, without installing or updating any packages or changing any environment variables.