Project 3: Application Security

This project is due on **18Apr2020** at **6 p.m.** and late submissions will be handled as discussed in the syllabus. Submissions will receive a zero (0) if submitted after **21Apr2020** at **6 p.m**. Please plan accordingly for your schedule and turn in your project early if appropriate.

The code and other answers you submit must be entirely your own work. 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. If you have questions about the meaning or scope of a question, post on the Canvas Discussion board. You may consult published references, provided that you appropriately cite them (e.g., with program comments).

Disclaimer

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 details of the target system, so we are providing a Kali Linux VM for you to develop and test your attacks. This VM is specially configured with certain features disabled to make the problems more deterministic and more straight-forward to exploit.

The VM must be setup correctly in order to ensure that your solutions work correctly when grading. It is your responsibility to ensure that your VM is setup correctly by following the below steps:

1. Download VirtualBox from https://www.virtualbox.org/ and install it on your computer. VirtualBox runs on Windows, Linux, and Mac OS.
2. Download the project VM's OVA from https://auburn.box.com/s/vcptkk6ldwmfc5tghju528l00wxma7qx. This file is moderately large (3GB) but can be re-used to create the VM from scratch as many times as needed.
3. Launch VirtualBox and select File ▷ Import Appliance to add the VM.
4. Start the VM and login with the username `kali` and password `kali`.
5. ***Do not* update the software in the VM.** Updates to the compiler or C libraries might change your solutions. We will grade using the same VM in its pre-installed state.

6. Navigate to `/home/kali/targets/`. This directory contains the code for all problems.
7. Run the command `./setcookie <your-auburn-email>`. Each student's targets and solutions will be slightly different, so make sure your email address is correct! Use your root email addres and *not* an alias (i.e. azs0249@auburn.edu vs. aaspring@auburn.edu)
8. Run `sudo make` to compile and configure the target binaries.

Resources and Guidelines

No attack tools allowed! Except where specifically noted, you may not use special-purpose tools meant for testing security or finding/exploiting vulnerabilities. You must complete the project using only general purpose tools, such as `gdb` and `objdump`.

GDB You will make use of the GDB debugger for dynamic analysis within the VM. Some useful starting commands are "`disassemble`", "`info`", "`x`", and "`stepi`". There is a quick-reference sheet in Canvas that will be helpful if you do not have experience with GDB.

x86 assembly There 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.

The VM's GDB installation is configured to use Intel syntax for disassembly. If you are more comfortable using AT&T syntax, remove the configuration line from `/home/kali/.gdbinit` to revert to the GDB default of AT&T.

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 with your cookie compiled as described above and executed within the given 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 an unintended 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 `buf` 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 *0x08XXXXXX
   ```

   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 (`#`) and running `whoami` will output "`root`". Running `exit` will return to your normal shell. It should be noted that when running inside GDB, it may spawn a non-root shell even if it would spawn a root shell outside of GDB. For grading, your solution will be run as described above and its operation while running inside GDB is irrelevant.

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 such that it is impossible to execute instructions stored on the stack. You can overflow the stack and modify values like the return address, but you can't inject and jump to shellcode like you could with previous targets. 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. There are simpler and more deterministic ways to solve this problem and environmental variable may not behave predictably during grading.

Bonus Targets

Undergrad Section: Targets 6, 7, and 8 are optional bonus problems worth 5, 10, and 10 points respectively. The maximum grade is 125.

Graduate Sections: While this entire project is optional for the graduate sections, you may complete it for bonus points on your overall grade (100 points max). If you choose to complete it, you *must* successfully solve targets 0-5 before attempting targets 6-8. If targets 0-5 are not correctly solved, targets 6-8 will not be graded. Correct solutions for targets 0-5 are worth one (1) bonus point total, target6 is worth one (1) bonus point, and targets 7/8 are worth two (2) bonus points each. This means you can add up to six (6) points to your overall grade through this project.

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, ASLR 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. You may **not** use any attack tools on this target.

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 solved this target correctly.

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 and the `ROPgadget` command is already installed on the VM. You may use any tool of your choosing to find ROP gadgets (or find them manually for added fun), but you may not use any tool or feature of a tool for a purpose other than locating ROP gadgets in the binary.

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)"
```

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

target8: Real-World Vulnerabilities (*Difficulty: Very Hard*)

Choose a library or application written in a memory unsafe language that is widely used in the real-world. Once chosen, identify a currently unknown binary vulnerability and create a local proof-of-concept exploit (PoC) for it. Once you have a working PoC, immediately contact the instructor and we will begin the Responsible Disclosure process. Under no circumstances should you exploit software on any system you do not own nor widely share the details of the vulnerability or PoC prior to disclosing. **If you have any questions about whether you should or should not do something, ask!**.

Tools: There are no restrictions on what tools may be used for this target. Code analysis, debuggers, fuzzers, and everything else is fair-game.

Advice:

1. Eschew well-known and well-studied applications such as Chrome and the Linux kernel. These are high-value targets that have large numbers of very experienced people spending large amounts of time searching for vulnerabilities.
2. Select an open-source library/application. While it is possible to find vulnerabilities in precompiled binaries, they are much more difficult to locate, understand, and PoC.
3. Complicated deserialization logic is often a fruitful source of binary vulnerabilities. Formats such as PDF, AVI, and SVG are complex and extremely difficult to implement safely and correctly in memory unsafe languages.
4. In my opinion, the best-case location to begin looking for a "hacker's first CVE" is network services in embedded systems libraries. Though not a guarantee by any measure, they are often as complex as general network services but do not have the same attention or infrastructure defenses built-in. (dnsmasq is a potential example)

What to submit You should not submit anything to Canvas for this target. Grading will be completed by the instructor based on the specific instance.

Submission Details

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.

You will submit your solutions (`solX.py`) the Project 3 Canvas assignment. Each solution should be reasonably commented to explain its operation and all solutions must use the filename listed above. Your submission must be as individual scripts and *not* as a tar or zip file.