Project 3: Binary Exploitation

This project is due on 02Nov2020 at **6 p.m.** and late submissions will be handled as discussed in the syllabus. Submissions will receive a zero (0) if submitted after 05Nov2020 at **6 p.m**. Please plan accordingly for your schedule and turn in your project early if appropriate.

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*.

Introduction

In this project, you will explore the world of binary exploitation through hands-on practice and experimentation. The targets are created specifically to add complexity from very-simple to relatively complex in an incremental manner. **If you do not understand why a solved-target works, to not progress to the next target!**. If you can not rationalize a behavior please trying to look-up the answer on the Internet but you are free to ask non-solution-specific questions[1] on Piazza or solution-specific questions[2] in Office Hours or over email (preferably Office Hours).

THIS PROJECT TAKES TIME AND YOU ARE UNLIKELY FINISH IF YOU START THE SATURDAY BEFORE IT'S DUE.

You may work on this project solo or in groups of no more than two. It is entirely your choice but the responsibility of forming, communicating, coordinating, and collaborating is yours as well. The groups may be across any sections of the course and are not restricted by in-person or distance-learning aspects. Piazza is an excellent way to find other people who wish to work in a group. In your submission to Canvas, you must include a `team-members.txt` file which lists them group members' names and their email addresses used to create the cookie **even if there is only one team member**.

[1] Examples: "What is get_pc_thunk.ax?", "Why do instruction addresses start with 0x08 and stack addresses start with 0xbf?", "How do I find the location of a function in memory?", etc.

[2] Example: "Why does it work when I pad with 14 bytes instead of 17?", "Why does the value 0xaabbccdd work but not the value 0xaabbccde?", etc.

Setup

The predictability of buffer-overflows and their exploitation depends on numerous details so we are providing a Kali Linux VM for you to develop and test your attacks in. This VM is specially configured with certain settings and features to make the problems more deterministic and more straight-forward to exploit. In order to ensure the reproducibility of your work, the VM must be setup correctly. **It is your responsibility to ensure that your VM is setup correctly.** The exact steps are below but you are welcome to use Piazza, Office Hours, or email if you are unsure of anything:

1. Download VirtualBox and install it on your computer. VirtualBox runs on Windows, Linux, and Mac OS.
2. Download the project VM's OVA from https://aub.ie/5370-proj3-vm. 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.** We will grade using the same VM image in its pre-installed state and local changes to the compiler, libraries, versions, etc may cause your solutions to work on your VM but not ours.
6. Navigate to `/home/kali/targets/`. This directory contains the code for all problems.
7. Run the following command with you and your partner's email address (if you are working solo, leave-off the second argument):
   `./setcookie <your-email-name>@auburn.edu <partner-email-name>@auburn.edu`
   Use your "root" address assigned to you and *not* an alias you chose. Each group's targets and solutions will be different and incorrectly creating your cookie and/or changing it after you've started working may cause your solutions to stop working. **If you have problems with this step, contact Dr. Springall as soon as possible to find a work-around.**
8. Run `sudo make` to compile and configure the target binaries.
9. Create a `team-members.txt` file with the names and email addresses of you and your partner (only yourself if solo). **INCLUDE THE SAME EMAIL(S) AS WHAT WAS USED TO CREATE THE COOKIE.** Failure to do so will result in your solutions not working when they are being graded.

Guidelines

NO ADVANCED TOOLING You may not use any special-purpose tools meant for testing binary security or finding/exploiting vulnerabilities. It is entirely possible to complete this project in the time available using only gdb, Python3, and a text editor. You may use other general-purpose tools such as xxd, hd, and objdump but they are not required and sometimes make things significantly harder. If you are in-doubt about a tool, ask on Piazza.

GDB You are expected to make heavy use of the GDB debugger for dynamic analysis within the VM. There are many, many resources available on the Internet if you have never used it before but the most helpful resources are the "cheat-sheets" of commands (personal favorite).

x86 assembly There are many good references for Intel assembly language but note that our project targets use the 32-bit x86 ISA. There are many similarities between the 32-bit and 64-bit versions but they **are not identical**. If you are reading any online documentation and it is not reflective of what you are seeing, you may be reading the 64-bit version or a completely different ISA such as ARM or SPARC. You should not need to revert to the Intel architecture manuals but they are the ground-truth for the x86 ISA.

The VM's GDB installation is pre-configured to use Intel syntax when showing instructions. If you are more comfortable using AT&T syntax, remove the configuration line from `/home/kali/.gdbinit` and restart GDB.

Do not modify the targets We will be using the targets as distributed and any change in source-code is likely to cause your solutions to not work correctly when grading. If you wish to play with variations either for fun or for testing, copy them to a different directory and keep them isolated from the original targets.

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.

Pro-Tips

- **START EARLY** — It takes time to solve some of the below targets especially if you are new to Linux, the terminal, or GDB.
- **Make your life easier** — Python3 has many built-in functions/libraries/syntaxes that can make your life significantly easier:
  - `b"\xYZ"` is quick and easy to to represent a byte.
  - `"a"*4` will create the string `"aaaa"` and it works the same for bytes.
  - `sys.stdout.buffer.write(aBytesVariable)` will print to `stdout` without trying to re-encode for the terminal.
  - `struct.pack("<I", 0x11223344)` is an easy way to turn a hex-value into a 32-bit little-endian int's bytes.
- **Solve one problem at a time** — Though this sounds obvious, it's actually the biggest thing that people struggle with. In general, you can follow the following steps to guide you through all of the targets:

1. Identify the weakness that you want to exploit.
2. Identify what you control and what you want to influence (logically and the actual memory addresses).
3. Build your exploit with an easy-to-see template such as "aaaa...aaaa" for padding, "bbbb...bbbb" for shellcode, "cccc" for an address, etc.
4. Run your templated exploit in GDB and make sure everything lines up the way you think it does.
5. Replace your template elements with the real elements.
6. Run in GDB and make sure everything still lines-up.
7. Run multiple times without GDB to ensure that it works and is consistent.

Part 1: Going Out-of-Bounds

For all of the below targets, your job is to exploit the weakness built into each target to print the string specified. It is important that make sure that the string outputted is the **exact format described**. There are no guarantees of partial-credit for any target output that is "close" but incorrect.

All of the targets for this portion accept input via `stdin`. You can develop/test your exploit however you like but your soution will be graded with the command:

```
python3 solX.py | ./targetX
```

target0: Overwriting a variable on the stack (*Difficulty: Easy*)

This target takes a string from `stdin` and uses it to give a user-specific output. Your job is to provide an input that causes the program to output: "`Hi` *`email-name`*`! Your grade is A+.`". To be clear, it should print that string **exactly** with only your email address's name replaced and not a slightly modified version of that string.

target1: Overwriting the return address (*Difficulty: Easy*)

This program takes input from `stdin`, ignores it, and calls one of two functions to print a static message. Your job is to provide input that changes the static message to a different one and outputs: "`Your grade is perfect.`". You should do this by hijacking the program's control-flow to make it execute a function which is available but never called in the current form.

Part 2: Popping Shells

The targets for this part of the project are owned by the `root` user and have the `suid` bit set. Your job is to exploit the weakness to cause a "root shell" to be launched[3]. Instead of passing your exploit to the target via `stdin`, you will pass your exploit as a command-line argument. The `shellcode.py`

[3]There are numerous ways to check if a spawned shell is a user-shell or a root-shell but the easiest is with the `whoami` Bash command. If the output is "root", you can be confident it is a root shell.

file on Canvas contains known-good shellcode to launch a shell. You are **highly** encouraged to use it but are not required to do so. To use it, you only have to place it in memory, cause the instruction pointer to begin executing at the beginning of the bytes, and it will spawn the root shell for you.

It is not important or necessary to understand how it works[4] but the mechanisms shown below will make it significantly easier to work through these targets. It will execute the command `python3 solX.py` and immediately call `targetX` with `argv[1]` set as the contents of the script's `stdout`. Your solution will be graded with the Bash version listed below:

Bash Command: `./targetX $(python3 solX.py)`
GDB Command: `run $(python3 solX.py)`

★★★★★ **WARNING** ★★★★★

Configuring executables similar to these targets means that even though a non-root user can execute them, the program will have all the abilities and permissions of the root user (why you are able to launch a root shell by correctly exploiting them). This is **an extremely dangerous thing to do** and is only done here for illustrative purposes. In the real-world, you should always explore if there are other ways that are safer and less likely to result in things going *horribly wrong*. There usually are and using the setuid+root is a cheap and lazy cop-out.

target2: Return-to-Shellcode (*Difficulty: Easy*)

In this target, you will use a standard return-to-shellcode attack. Your first goal should be determining a mechanism to inject your shellcode and then finding placing it in the program's memory in a predictable location. After that, you should exploit the standard buffer overflow to execute your shellcode (DEP is disabled).

target3: Overwriting the return address indirectly (*Difficulty: Medium*)

This target is similar to the previous, but the buffer overflow is heavily restricted. Because of this, you are not able to directly overwrite the return address and will have to find another way. Just like target2, DEP has been disabled.

Part 3: The Fun Part

Similar to Part 2, the targets for this part are owned by the `root` user and have the `suid` bit set and your job is to exploit the weakness to cause a "root shell" to be launched. The difference is that you must use more complicated tools in your binary exploitation toolkit. The concepts are the same but the techniques are different. Though still far from the complexity associated with real-world binary exploitation, these are starting to add complexity and protections in the vein of real-world defenses.

target4: Beyond strings (*Difficulty: Medium*)

This target takes a filename as a command-line argument and parses that file to load it into memory. The file format is a 32-bit count followed by that many 32-bit integers (all little endian). Your goal is to figure out how to construct an input file that will cause the target to spawn a root shell.

[4] It is a cool party-trick though.

DO NOT SUBMIT THE INPUT FILE TO CANVAS. You should submit a short Python3 script which will output the malicious input file's contents to `stdout`. This output will be redirected to a local file when grading and that file will be used for testing. The command used for grading will be:

```
python3 sol4.py > tmp; ./target4 tmp
```

target5: Bypassing DEP

(*Difficulty: Hard*)

This program resembles `target2`, but it has been compiled with DEP enabled such that it is impossible to execute shellcode stored on the stack. You can overflow the stack and modify values including the return address but you can't transfer execution to any data which you injected. You need to find another way to spawn a shell. To make things less painful, it is perfectly acceptable for target5 to segfault **after** the root shell has been closed.

Do not submit a solution that depends on local environment variables! There are much simpler and more deterministic ways to solve this problem and any solution which uses environmental variables in the commonly-exploited way will be marked as incorrect. The command for grading will be:

```
./target5 $(python3 sol5.py)
```

Submission Details

Various expectations of your Canvas submission are listed below. **They are non-negotiable!** If you submission fails to meet these expectations, you may receive a zero (0) for the entire project. If you have any questions, about submission format, details, contents, or anything discussed below, ask on Piazza. If you believe there is an error in the below requirements, contact Dr. Springall as soon as possible.

File uploaded to Canvas:

- It must be named `project-3.tar.gz`.
  - Canvas will auto-rename it but this should be the name of the file you upload to Canvas.
- It must contain:
  - `sol[0-5].py` (i.e. your solutions)
  - The `cookie` file generated by the `setcookie` script.
  - The `shellcode.py` script.
  - The `team-members.txt` file containing the names and emails of the team members you worked in a group with. **This file must be included even if you worked solo.**
- The above files must be in the root of the tarball and not nested in a directory.
- It must not contain any file other than those listed above including, but not limited to binaries, targets' source code, the input file for target4, ...

  - You can list the structure and contents of a tarball with the Bash command:
    `tar -tf project-3.tar.gz`

- It must be a gzip'd tarball
  - Zipfiles are **not** acceptable.

The easiest way to create your submission file is with the Bash command:
`tar -czf project-3.tar.gz cookie team-members.txt shellcode.py sol[0-5].py`

Specific to each target's solution:

- It must be a Python3 script.
  - Python2 is dangerous and wholly unacceptable.
- It must be organized, cleanly written, and the logic must be explicitly clear from the source.
  - Either use descriptive variable names and combine them in-order when printing or add comments describing what is happening.
  - A solution of the form below listed below is unreadable, unacceptable, and will be marked as incorrect regardless of its behavior.
    "`print("a8\a2XAAAx9jd8g fs& 235\034")`
- With the exception of importing the `shellcode.py`, it must not use any Python package that is not in the Python3 standard library.
  - It is important to note that the Kali VM has a large amount of software installed outside of the base-distribution and standard libraries. Just because you didn't install it doesn't mean it's in the Python standard library.
- It must not error when using the appropriate command for grading
  - If the solution exits successfully but the target crashes, you may receive partial credit.
- It must not do anything other than write to `stdout` including, but not limited to:
  - Writing directly to a file
  - Reading anything including files, variables, configurations, etc
  - Making network requests
  - Attempting to install packages
- It must not output any debugging information or strings/bytes/letters/symbols/. . . not part of your payload/exploit.
- Where appropriate, it must output **your** email address and not that of your partner's.