Project 2: Binary Exploitation

This project is due **Sunday, 13Oct2024 at 6pm CT** and must be submitted through the Canvas assignment (if early/on-time) or by emailing the TA (if late). Late assignments will be penalized as described in the syllabus.

Disclaimer

This project asks you to develop binary exploits and test them in a virtual machine you control. Attempting the similar 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 attempt to exploit 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 for this purpose from very-simple to relatively complex in an incremental manner. **If you do not understand why a solved-target works, the next target may require you to build upon your assumed understanding.** If you have thoroughly investigated and still can not rationalize some behavior, you are welcome to ask for clarification from the instructor or TA.

THIS PROJECT TAKES TIME AND YOU ARE UNLIKELY FINISH IF YOU START TWO DAYS BEFORE THE DEADLINE.

Setup

The predictability of buffer-overflows and their exploitation depends on numerous details so we are providing a Debian Linux VM for you to develop and test your attacks in. This VM is specially configured with certain settings and features to make the targets and required solutions more deterministic and straight-forward. In order to ensure the reproducibility of your work, the VM's configuration, OS's settings, and installed software should only be changed after careful evaluation of its impact. The exact steps are below and you are welcome to ask for clarifications but **it is your responsibility** to ensure that the VM you are using locally is setup correctly.

1. Download VirtualBox (https://virtualbox.org) and install it on your computer. VirtualBox runs on Windows, Linux, and macOS-Intel. If you only have ARM hardware available (notably the M1/M2 family of macOS devices but Windows devices are not unheard of), please contact Dr. Springall as soon as possible and a work-around will be provided.
2. Download the project VM's OVA (link on website). This file is moderately large (2+ GB) but can be re-used to create new VMs 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 `student` and password `Auburn`[1].
5. ***Do not* update the software in the VM.** We will grade using the same VM image in its existing state and local changes to the compiler, libraries, versions, etc may cause your solutions to work in your VM but not ours.
6. Open a terminal and navigate to the directory containing the code and scripts for the project:
   ```
   /home/student/targets/
   ```
7. Run the below command with your **email address**. Use your "root" address assigned to you and *not* an alias you chose. Each student'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.**
   ```
   ./setcookie <your-email-name>@auburn.edu
   ```
8. Run the below command to compile and configure the target binaries:
   ```
   sudo make
   ```
   It is **OK** for this command to printer warning messages as long as it still outputs all binaries.

[1]If needed, the root password is also `Auburn`.

Targets

The target programs for this project are short, straight-forward C programs with mostly-clear vulnerabilities. Source code is provided 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.

Part 1: Get 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.

target0: Overwriting a variable

Required — 5370 & 6370
Bonus — N/A

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. Of note, it *should not* output second-half of your email address (`@auburn.edu`), only the first-half.

Command used for grading:

```
student$> python3 sol0.py | ./target0
```

target1: Overwriting the return address

Required — 5370 & 6370
Bonus — N/A

This program takes input from `stdin`, ignores it, and calls one of two functions to print a 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 executes a function which is available but never called in the current form.

Command used for grading:

```
student$> python3 sol1.py | ./target1
```

*: https://www.youtube.com/watch?v=odZtFcRNdes

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 provided `shellcode.py` Python3 module 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.

★★★★★ WARNING ★★★★★

Configuring executables similar to these targets (with the `setuid` bit set) 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 for illustrative purposes. In the real-world, you should always explore other ways that are safer and less likely to result in things going horribly wrong.

target2: Return-to-Shellcode

Required — 5370 & 6370
Bonus — N/A

In this target, you will perform a standard return-to-shellcode attack. Your first goal should be determining a mechanism to inject your shellcode 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).

Command used for grading:

```
student$> ./target2 $(python3 sol2.py)
```

target3: Bypassing DEP

Required — 5370 & 6370
Bonus — N/A

This program resembles `target2`, but it has been compiled with DEP enabled so 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 this target to segfault **after** the root shell has been closed.

Do not submit a solution that depends on local environment variables not set in the OVA! There are much simpler and more deterministic ways to solve this problem and any solution which uses environmental variables in this way will be marked as incorrect.

[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.

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

Command used for grading:

```
student$> ./target3 $(python3 sol3.py)
```

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 the more complex techniques from lecture. The concepts are identical to Part 1 and 2 but the mechanisms are different. Though still far from the complexity associated with real-world binary exploitation, these are starting to add complexity and protections that begin to approximate the real-world.

target4: Overwriting the return address indirectly

Required — 6370
Bonus — 5370

This target is similar to target2, 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.

Command used for grading:

```
student$> ./target4 $(python3 sol4.py)
```

target5: Beyond strings

Required — 6370
Bonus — 5370

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.

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.

Command used for grading:

```
student$> python3 sol5.py > tmp && ./target5 tmp
```

Submission Details

Various expectations of your Canvas submission are listed below. **They are non-negotiable!** If your 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, please ask.

File uploaded to Canvas:

- Your tarball (do not submit a zip-file) must contain:
  - `sol[0-5].py` (i.e. your solutions)
  - The `cookie` file generated by the `setcookie` script.
  - An ascii-only `setup.txt` file with a minimum of your full AU email address used to create the cookie. This file should also contain a *small* amount of text describing how to build your code if you are not using Python3.
- 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, input file for target5, ...
- If you are in the 5370 section and decide not to attempt the bonus, include a Python3 script with the correct name but only the line:

  ```
  sys.exit(0)
  ```
- You **may not** upload a zip-file.

The easiest way to create a tarball to submit is with the Bash command:

```
tar -cf project-2.tar cookie sol{0,1,2,3,4,5}.py
```

Specific to each target's solution:

- If using Python, you must use Python3
  - 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 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 provide `shellcode.py`, it must not use any Python package that is not in the default Python3 interpreter
- 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.

Guidelines & Pro-Tips

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.

START EARLY It takes time to solve some of the below targets especially if you are new to Linux, the terminal, or 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 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/student/.gdbinit` and restart GDB.

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. Some commands that will be useful during your journey:

Display register value — `(gdb) info reg` <*register-name*>

Display the address of a function — `(gdb) info addr` <*func-name*>

Display the top stack-value as hex — `(gdb) x/1wx $esp`

Pause after the prolog of a function next time it's called — `(gdb) break` <*func-name*>

Pause before executing a specific instruction — `(gdb) break *0x`<*instruction-addr*>

Set cmd-argument from solution and restart — `(gdb) run $(python3 solX.py)`

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.

Make your life easier Python3 has many built-in functions/libraries/syntaxes that can make your life significantly easier:

- `b"\x`YZ`"` is quick and easy to to represent a byte.
- `b"a"*4` will create a Python string containing four 0x61 bytes.
- `sys.stdout.buffer.write(aBytesVariable)` will print to `stdout` and will avoid many different issues related to encoding.
- `struct.pack("<I", 0x11223344)` is an easy way to turn a hex-value into a 32-bit little-endian int's bytes.
- Use the shortcut for appending bytes (`var += val`)

Solve one problem at a time Though this sounds obvious, it is almost always the biggest problem that students struggle with. In general, you can follow the following steps to guide you through all of the targets:

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

Python3 is your Friend The actual code required to complete this project is extremely small and relatively simple. You are welcome to use whatever language you like but Python3 is *highly recommended* as a well-engineered and readable solution script is 5–10 lines. Though not required to be of this format, the instructor's solution have the general template of:

```
import sys
import struct
from shellcode import shellcode

# Define some constants
# Do some math

to_be_outputted = b''
# Add some things to to_be_outputted
# Add some padding to to_be_outputted
# Add some more things to to_be_outputted
sys.stdout.buffer.write(to_be_outputted)
```

Common Problems/Questions

VM Window Too Big If your local setup does not show the entire VM window to the extent that you have to scroll-around, you can reduce the window-size by:

1. Right-click the VM in the Oracle VM VirtualBox Manager window
2. Select “Settings”
3. On the left-side or top (depending on OS), select Display
4. On the right-side, change Scale Factor to “100%”

VM Window Still Too Big If the above window is still to big, you can further reduce it by:

1. Boot the VM and login
2. Inside the VM, right-click the desktop background
3. Select “Display Settings”
4. Set Resolution to 800x600 (4:3)
5. Click “Apply” in the top-right corner of the window (you may need to scroll to see it.
6. Click “Keep Changes”
7. Reboot

VM Window Size (other) You can use combinations of both VirtualBox settings (first approach above) and Debian settings (second approach above) to attempt to find more-usable configuration. Neither of these settings affect the OS configuration to the degree that your solutions will not work on the auto-grader VM but you can always double-check.

When Starting VM for First Time, It Crashes Immediately Though the error messages are often of no-help, this is often due to yet another graphics-driver but related to hardware acceleration. The below steps but if the VM continues to crash, contact the instructor/TA as soon as possible.

1. Right-click the VM in the Oracle VM VirtualBox Manager window
2. Select “Settings”
3. On the left-side or top (depending on OS), select Display
4. Towards the bottom, you should see “Extended Features:” with a checkbox labeled “Enable 3D Acceleration”
5. Invert that setting (not always on/off by default)

The VM is Incredibly Slow Due to virtualization, the VM *will be slower than your normal computer*. If it is too slow to work effectively, there are things you can do to try and make it run faster without affecting deterministic grading:

- Shutdown and then re-boot the VM
- Close any other running VM
- Close other applications running on your bare-metal OS (Chrome is often the culprit)
- Increase the virtual CPU and/or RAM via Settings ▷ System (OVA is set to 2 cores, 2048MB RAM)
- Reduce VirtualBox’s graphical scaling (see above)
- Reduce the VM’s resolution (see above)
- Reboot your bare-metal OS (rarely needed but has worked in the past)

Starting-Over No matter what, it is always possible to delete the existing VM (right-click, Remove, Remove All Files) and then re-import the OVA to create a fresh VM without any software additions, configuration changes, etc. The down-side is that A) you’ll have to setup your cookie and re-compile the target binaries and B) any files on the old VM such as a finished or in-progress solution **will not be transferred to the new VM**. You are *highly encouraged* to ensure you have some form of backup whether it’s in a private Github repo or just a picture of the script on your phone.

Solution Works Inside GDB but Not Outside It Although not always the case, GDB’s environment changes sometimes cause memory addresses to similar but not identical (4–16 bytes difference usually). You can either A) attempt to recover the new address via a “core dump” file[5] or simply modify your solution’s addresses blindly to see if you can stumble upon it easily[6].

[5]Requires non-trivial but safe OS configuration changes which are widely documented.

[6]From experience, can usually be found in a few minutes of trial-and-error by incrementing and decrementing.

Opening a Terminal After logging in, click on the black-square icon at the bottom of the screen to open a shell. If you would like to open a second shell or you closed your first one, click “Activities” in the top-left of the screen and the black-square will be available at the bottom of the screen again.

Can’t Access the Internet The network adapter has been intentionally disabled in the OVA distributed to prevent accidental software updates, installations, and the like. **You can re-enable if you’d like** with the command:

```
sudo ifconfig enp0s3 up
```

Using Copy-Paste Copy-pasting in Linux terminals usually works very differently than Windows/macOS due to the fact that Ctrl+c means “send SIGINT to the running process”[7]. This is highly useful when you have a miss-behaving process but becomes annoying when you forget. To copy-paste as normal, use Ctrl+Shift+c and Ctrl+Shift+v.

Installing (anything) You are welcome to install any software/library/application/etc you wish other than “advanced tooling” as described above. That being said, you should be aware that installing software can often update shared dependencies which can cause your memory addresses, offsets, output binaries, etc to be different (i.e will not work correctly on the auto-grader). If you do install any software (no matter how seemingly trivial) you should be sure to test your solutions in the distributed OVA without any additional software installed.

VirtualBox Additions Many students find it extremely helpful to install “VirtualBox Additions” to make working inside the VM simpler. You are welcome to do so if you like but A) this is not a requirement and B) this counts as installing software per above. If you do install, be sure to double-check with the distributed OVA directly as the VirtualBox Additions modifies the OS in a way may be specific to your laptop.

VMWare Player/Workstation/Fusion, Parallels, and Other Hypervisors Unfortunately, none of these appear to be compatible with VirtualBox-generated OVAs. Due to the fact that VirtualBox is A) free without requesting a license, B) available on all three major OSes, and C) has a history of “playing nice” with the other hypervisors. . . **only VirtualBox will be fully-supported by the instructor and TA**. If you find a way to import/run the distributed OVA but later run into an issue with it, we will most likely tell you to install VirtualBox and see if you have the same problem.

Other OSes and Not-From-OVA VMs It is an exceptionally poor idea to copy the target binaries/source to any VM (all VMs created from the distributed OVA already have them) whether or not it is running the same version of Debian as the distributed OVA VMs. It is a near-certainty that your solutions will not be correct when run by the auto-grader VM.

Errata

- 04Oct2024 — Remove reference to “Part A”

[7] Ctrl+v is similarly different