15/18-330 F22: Introduction to Computer Security
Evil Corps Exploitation Exercise

September 20, 2022

1 Introduction

As a special agent from the `330` hacking group, you have been assigned a secret mission. In this mission, you need to break into the `Evil Corps` systems and perform some secret operations using the company's software in environments where ASLR may be either disabled or enabled. The tasks are ordered, in this document, approximately by increasing difficulty. Each homework assignment will ask you to complete one or more of these tasks.

Please read the entire document before starting, including the tips section at the bottom.

1.1 Grading

1.1.1 Rules

- You can discuss concepts with others, but you **must create your own** exploits.
- Use Piazza to ask questions.
- **DO NOT** try anything that you suspect could cause problems for the grading infrastructure.
- **DO NOT** try to circumvent the autograder functionality.
- **DO NOT** try to obtain root access on the systems.
- **DO NOT** brute-force the autograder to find a working exploit.

1.1.2 Criteria

Each homework will specify whether a task will be graded purely based on successfully exploiting the software, or whether it will require additional details written up. For the latter, the grading will be based upon the following 3 criteria. Partial credit will be given as deemed appropriate by the graders. See the end of the writeup for answers to a sample problem.

- **Reversing (4 pts):** Explain what the program does, especially how the *user input* is processed (feel free to present a pseudocode version of the program). This should include enough detail that the reader could reimplement the program if needed, just from your description.
  Then, identify the *vulnerability* in the program, i.e., which kind of vulnerability it is, where the vulnerability is, and why it is in fact a vulnerability. Be sure to explicitly name the functions relevant to the vulnerability (*strcpy*, *printf*, etc.). If a problem asks questions specific to the task, your answer to those questions must be included in this section.
- **Working exploit (15 pts):** Create *your own exploit* that successfully works against the provided binary on Autolab.
- **Exploit details (6 pts):**
  - **Explanation (2 pts):** Provide a full description of how your exploit works. Be sure to explain how your exploit gets around any countermeasures in the target program and the operating system.
  - **Stack diagram (4 pts):** Provide a thorough *stack diagram*. We will accept neatly hand-drawn diagrams, spreadsheet screenshots, LaTeX tables, MS Paint diagrams, or any other sufficiently neat and labeled visuals. Stack diagrams will be graded using the following rubric.
    1. If the function that contains the vulnerability is `foo`, then the stack diagram must, at the very least, include full details of everything on `foo`'s stack frame when the exploit begins.

2. If other portions of the stack are relevant to the exploit, be sure to show those details too.
3. As shown in the example at the end of this document, each stack slot in your diagram should represent a 64-bit value, and each slot that contains important data should have an associated address label.
4. Information that has semantic information (e.g., "user's phone number") should be labeled as such, rather than just using the register name it happened to be held in.

1.1.3 Working on Exploits

You must work with the binaries provided to you in *exploit-files.tar* provided on Autolab. DO NOT recompile any binaries for which source code is also provided; you must work only with the provided binary. For all five tasks, we strongly recommend that you work on your exploits on `linux.andrew.cmu.edu` or at least make sure they work on `linux.andrew.cmu.edu` so that you are working in the environment that is the closest to Autolab. You can log into the Linux server by running `ssh <andrewID>@linux.andrew.cmu.edu` and entering your Andrew password.

- **Tasks Alice, Bob, and Dante:** Your exploits for these tasks are meant to work in an environment where ASLR is disabled. Note that you do not need to do anything special to set up this environment; we do not use OS support to disable ASLR but rather we compile the binaries to have fixed stack addresses to simulate disabling ASLR.
- **Tasks Christa and Elaine:** Your exploits for these tasks are meant to work in an environment where ASLR is enabled. Note that you do not need to do anything special to set up this environment; ASLR is enabled by default on most operating systems.

1.1.4 Submission

Your submission will typically contain two parts:

- **To Gradescope**: A typed *writeup* in PDF format describing the three above criteria from Section 1.1.2 for every task. A LaTeXtemplate is provided to you. As part of submitting your PDF to Gradescope, you will be asked to specify on Gradescope which page of your PDF corresponds to which problem of the assignment.
- **To Autolab**: A TAR archive of a folder with all your solution exploits called *submission.tar*. Create a folder named *submission/* Inside this folder, include a separate text file for each exploit for each problem labeled {*problem*}*.txt*, e.g., *alice.txt*. If your exploit depends on other files, you must ensure the name of the task is the first substring of the file name and include any such helper files in the tarball. For example, if you need to include a python helper for Alice's task, some acceptable file names are "alice.py" or "alice-helper.py". Your folder may have as many exploits as problems you have solved; if you do not submit an exploit for a problem, you will simply receive 0 on that problem for that attempt on Autolab. Your exploit will be tested from the command line in Autolab, *not* from within `gdb`. You can create your tar archive for submission by running: `tar cf submission.tar submission/`

Note about paths: If by any chance your exploits require the full path of the binary to run correctly, replace the path leading up-to and including the `problems` directory with {`0`}, e.g., `/path/to/problems/bob/snote` should become {`0`}`/bob/snote` in the submitted exploit text file. The autograder will replace the path with its own path to the binary. If you find that your exploits are working on `linux.andrew.cmu.edu` but not on Autolab, one possibility could be because of a mistake in setting the full path in your submission files.

2 Tasks

2.1 Task 1: `alice`

Your first task is to infiltrate the `Evil Corps` company's network. By eavesdropping on conversations between employees of the `Evil Corps` company, you have obtained access to a password recovery service. Your goal is to retrieve the credentials by exploiting the password recovery service (or maybe by guessing a correct PIN...). You should analyze and understand the binary `recoverpw` first, then find the vulnerability, and make an exploit for it. A successful attack will print out the recovered password without requiring authentication. (*Hint:* you might be interested in the `recover_passwd` function.)

2.2 Task 2: `bob`

Bob (Chief Technology Officer/CTO) of `Evil Corps` uses a program called `snote` (secure note) to record confidential messages with a buffer-overflow checking routine. Your goal is to exploit `bob/snote` to spawn a shell. (*Hint:* you might be interested in the `note` function.)

Note: For spawning a shell, while you could technically use any shellcode with your exploits, we **require** that you use the following for this task and subsequent tasks where you need shell code (note: not all subsequent tasks necessarily need shell code): `http://shell-storm.org/shellcode/files/shellcode-806.php`.

2.3 Task 3: `christa`

Christa (Chief Executive Officer/CEO) of `Evil Corps` is worried about hacking and has enabled Address Space Layout Randomization (ASLR). Your goal is to spawn a shell by exploiting `christa/snote`.

NOTE 1: For the **reversing** writeup criterion, describe why your exploit for `bob/snote` does not work for `christa/snote`. Explain the purpose of ASLR and its effect on achieving a successful exploit.

NOTE 2: Your exploits for `bob/snote` and `christa/snote` should be *different*. The main difference would be figuring out stack addresses by using `gdb` vs. bypassing ASLR altogether. **DO NOT** brute-force the system.

2.4 Task 4: `dante`

Dante (Chief Financial Officer/CFO) of `Evil Corps`, again, uses a program called `snote` to record confidential messages. Unfortunately, after Christa's computer was mysteriously hacked, Evil Corps has increased security to the point where you were unable to obtain a complete binary for Dante's `snote`. However, using some partial information disclosures, you were able to determine some crucial addresses within the binary.

1. `0x40119e:  pop %rax; pop %rdx; ret`
2. `0x4011a1:  mov %rdi, %rsi; ret`
3. `0x4011a5:  mov (%rsp), %rdi; add $0x8, %rsp; ret`
4. `0x4011ae:  mov %rax, 0x10(%rsp); ret`
5. `0x4011b4:  ret`
6. `0x4011b8:  <get_token>`
7. `0x4011e8:  <become_admin>`

You also managed to install a helper program, such that you can provide exactly 12 values that will be placed on `snote`'s stack as follows, just before `snote` executes a `pop %rbp; ret` sequence.

```
----------------------
|                    |  <-- arg12
----------------------
|                    |  <-- arg11
----------------------
|                    |  <-- arg10
----------------------
|                    |  <-- arg9
----------------------
|                    |  <-- arg8
----------------------
|                    |  <-- arg7
----------------------
|                    |  <-- arg6
----------------------
|                    |  <-- arg5
----------------------
|                    |  <-- arg4
----------------------
|                    |  <-- arg3
----------------------
|                    |  <-- arg2
----------------------
|  return address    |  <-- arg1
----------------------
|  old(rbp)          |
----------------------  <-- rbp, rsp
```

Your goal is to use your helper program to take control of Dante's computer by invoking:
`become_admin(get_token(0, 0xaaaaaa))`. In other words, you need to pass those specific arguments to the `get_token` function and then pass the return value of `get_token` to `become_admin`. To use your helper program, you should submit a file `dante.txt` that contains one line with 12 space-separated unsigned decimal numbers. If your attack is successful, `become_admin` will report that you have become an admin.

2.5 Task 5: `elaine`

Elaine the Chairwoman uses a program called `snote` to record confidential messages. Your goal is to, again, spawn a shell by exploiting `elaine/snote`.

NOTE: For the **reversing** writeup criterion, explain the purpose of a non-executable stack and its effect on achieving a successful exploit. Also, explain whether this binary is vulnerable to a stack-based buffer overflow.

3 Tips

1. You can print hexadecimal values using `perl` or `python` (it **must** be `python2` or you may have issues). For example:

```
perl -e 'print "\xFF\xFF"x4'
python2 -c 'print("\xFF\xFF"*4)'
```

2. How would you print a non-ASCII character? The character "A" is the byte 0x41, but we can just type "A". But what would we do for any bytes that we can't type? (Check out the ASCII table - only 0x32-0x7e are the majority of characters we can type with a keyboard.)

3. How do you give non-ASCII characters to an executable?

   Through stdin:

```
(python2 -c 'print "\x41"') | ./binary
```

   Through arguments:

```
./binary $(python2 -c 'print "\x41"')
```

   If your python command is printing tabs/spaces, make sure to enclose the argument as follows so that it's included in the argument (Important for Elaine's snote):

```
./binary "$(python2 -c 'print "\x41"')"
```

4. Make sure you close all your parenthesis/quotes. Make sure you use quotes in all the right places just like the previous tip.

5. Don't forget about the endianness of memory addresses.

6. Due to the environment variables, the stack addresses when a program runs in `gdb` may be different from those when the program is run directly. To mitigate this problem, run the following commands in `gdb` before running the program:

```
unset env COLUMNS
unset env LINES
unset env OLDPWD
set env _ $HOME
```

   Then, use *the absolute path* when running at the command line.

7. You can find various shellcode for Linux at `http://www.shell-storm.org/shellcode`. We recommend using `http://shell-storm.org/shellcode/files/shellcode-806.php`

8. Environment variables related to the name of the user or even the path to the current working directly can cause an exploit to be unreliable. In addition, stack addresses might not be exactly the same across different machines (for example, Autolab vs. the Andrew servers). One technique for building a robust exploit is to use a series of NOP instructions (`0x90`) to form a NOP slide, as discussed in lecture: `http://en.wikipedia.org/wiki/NOP_slide`.

9. You will never need to be doing over-complicated ROP gadget chaining. The most complicated gadgets you will be looking for are similar to what was presented in class (ret2ret/ret2pop). If you are having trouble constructing a solution, figure out whether you can use one or more gadgets of rets or pop(s)+ret.

10. The Global Offset Table, which contains the addresses of dynamically linked functions, is not placed in a randomized location by ASLR and can be viewed with `readelf -r`.

11. The assembly command `repnz scas` is just an efficient way to calculate string length.

12. The short write variation of a format string attack drastically reduces the amount printed to standard output. See Section 4.1, `https://cs155.stanford.edu/papers/formatstring-1.2.pdf`.

13. **Binary Hacking Course by LiveOverflow:** an amazing series of in-depth and from-the-bottom-up videos on how to do binary exploitation (seriously this guy's videos are great). Here are the videos we'd recommend checking out:
    1. 0x04: a good intro/review to x86 assembly
    2. 0x05/0x06: some info & tools on reversing binaries
    3. 0x0c/0x0d/0x0e/0x0f/0x11/0x13: a few videos that work through examples of types of exploit techniques you saw in class
    4. 0x12: a bit about the GOT/PLT

14. This is a wonderful resource for conceptually understanding format string attacks, even though it is written for 32-bit exploits:
    `http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html`.

    Some differences between 32-bit and 64-bit to watch out for:

    (a) 32-bit environments use 4-byte addresses, while 64-bit environments use 8-byte addresses. Just remember to keep this in mind when reading through the link.

    (b) In 64-bit environments, `printf` will first use `rdi` as the string argument, then search through `rsi, rdx, rcx, r8, r9` before looking through the stack for arguments to fulfill the format specifiers.

    (c) `%hn` format specifier writes two bytes to the specified memory address, while `%hhn` format specifier writes one byte to the specified memory address

15. **Hints for Elaine:**

    (a) Program arguments are stored contiguously on the stack, separated by the null byte that ends each argument string. I wonder what happens if we give an empty argument "".

    (b) See the assignments page on the course website for more thoughts on command-line arguments.

    (c) For debugging, remember to `set disable-randomization on` after executing gdb to ensure that your exploit works despite stack randomization.

Beyond the assignment: For this assignment, we do not deal with different users and permissions and just require your exploit to spawn a shell. However, with a real vulnerable system or in capture the flag (CTF) games, exploiting the vulnerable program to open a shell can give the attacker a privileged shell and access to protected data, if the software they are exploiting is not owned by them.

4 Sample Problem and Answers

The binary:

```
000000000040059d <call_me_maybe>:
 push   %rbp
 mov    %rsp,%rbp
 sub    $0x10,%rsp
 movl   $0x0,-0x8(%rbp)
 mov    $0x4006a0,%edi
 callq  400470 <puts@plt>
 mov    -0x4(%rbp),%eax
 leaveq
 retq

00000000004005bb <main>:
 push   %rbp
 mov    %rsp,%rbp
 sub    $0x20,%rsp
 lea    -0x18(%rbp),%rdx
 lea    -0x10(%rbp),%rax
 mov    %rax,%rsi
 mov    $0x4006b4,%edi    ; Pointer to "%d %d"
 mov    $0x0,%eax
 callq  4004a0 <__isoc99_scanf@plt>
 mov    %eax,-0x8(%rbp)
 cmpl   $0x1,-0x8(%rbp)
 jg     4005f7 <main+0x3c>
 mov    $0x4006c0,%edi
 callq  400470 <puts@plt>
 mov    $0x0,%eax
 jmp    400601 <main+0x46>
 callq  40059d <call_me_maybe>
 mov    $0x0,%eax
 leaveq
 retq
```

4.1 Example Stack Diagram

```
================================ <- main stack frame
| saved rbp                    |
-------------------------------- rbp
| scanf return value           |
-------------------------------- rbp - 8
| stack variable: user input 1 |
-------------------------------- rbp - 16
| stack variable: user input 2 |
-------------------------------- rbp - 24
|                              |
-------------------------------- rbp - 32
| saved return address         |
================================ <- call_me_maybe() stack frame
| saved rbp                    |
-------------------------------- rbp
| stack variable = 0           |
-------------------------------- rbp - 8
|                              |
|                              |
|                              |
|                              |
================================ rbp - 16 = rsp
```

4.2 What Does this Program Do?

This program checks that the user inputs two valid integers, separated by a space. If the input is invalid, it prints something (presumably an error message) and then exits. Otherwise, main will call `call_me_maybe`, which in turn prints something.

An example of a reasonable trace: User inputs two numbers separated by a space in main. The program confirms the input is valid and calls `call_me_maybe`. `Call_me_maybe` will print before returning to the caller, `main`. The program exits gracefully.