Buffer

Tuesday, October 11, 2022 at 9:00pm ET

1 Introduction

The remaining survivors of the latest Doeppner attack have found shelter in an abandoned fort. However, your team is running low on supplies, so you must lead a small raiding crew to venture out in your armored vehicle in search of more.

The armored vehicle's GPS system is completely computerized and relies on extensive knowledge of C and x86-64 assembly. In the following assignment, you will complete various tasks to avoid Doeppner-zombies on your supply run. Good luck!

2 Assignment

If this is your first time installing an assignment, or you need a refresher, please refer to the Github & Gradescope guide. This will walk you through how to get setup with Git and install an assignment.

The Github assignment link can be found **here**. Be sure to make commits frequently so you won't lose your work!

This assignment will help you develop a detailed understanding of x86-64 calling conventions and stack organization. It involves applying a series of *buffer overflow attacks,* one of the methods commonly used to exploit security weaknesses in operating systems and network servers, on an executable file called **`buffer`**. Our purpose is to help you learn about the runtime operation of programs and to understand the nature of this form of security weakness so that you can avoid it when you write system code. *We do not condone the use of this or any other form of attack to gain unauthorized access to any system resources.*

This assignment contains the following files:

- **`buffer`**: The buffer program you will attack.
- **`makecookie`**: Generates a "cookie" unique to you that's stored in cookie.txt
- **`hex2raw`**: A utility to help convert between string formats.
- **`README.md`**: The file where you will write your explanations of your exploits.

Your task during this assignment is to create buffer overflow attacks to cause the **`buffer`** program to behave in ways that help your raiding crew obtain supplies for your camp and return safely!

Note: Please make sure that you are running your code in a *Linux environment*: either a VM or using a department machine (FastX, ssh, in-person, etc.).

2.1 Collaboration

For this assignment:

- You may discuss the stack conceptually — what happens during function calls, how `%rbp`, `%rip`, and `%rsp` change, etc. — with other students in this course.
- You may not share any other aspect of your exploits (the txt file, assembly code) with other students in this course. Only you or a member of the course staff may examine any aspect of your exploits.

2.2 Hours Policy

We will have regular code hours for Buffer! Note, however, that we will be shifting some code hours to conceptual hours, in order to emphasize how important it is to conceptually understand the stack. Please check the calendar for a more up-to-date schedule of TA hours.

Please read through the Buffer Explanation handout before proceeding to the next section.

The Buffer Explanation handout contains important information about the stack, cookies, and the buffer program itself that you will need to write your exploit files. Please be sure that you understand these concepts before attempting the assignment!

3 Exploit Strings

To create your buffer attack, you will be writing your exploit strings into a .txt file. Your strings must be hex-formatted, meaning that each byte value is represented by two hex digits. The hex characters that you write in your .txt exploit files should be separated by whitespace (blanks or newlines). You can create one of these files for each of the buffer levels in your terminal by typing:

```
touch <name_of_level>.txt
```

What we write in our .txt files are hex-formatted strings. The program, however, expects a raw exploit string. Raw exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters, so we cannot use our .txt files directly. The program **`hex2raw`** will help you convert your .txt file exploit strings into a raw exploit string. It takes as input a .txt file containing a hex-formatted string and will output a .txt file containing the raw exploit string. You may call **`hex2raw`** using I/O redirection, like so:

```
./hex2raw < exploit.txt > exploit-raw.txt
```

`hex2raw` also supports C-style block comments, so you can mark off sections of your exploit string. For example, in your exploit.txt files, you may have:

```
bf 66 7b 32 78 /* address to some function */
00 00 00 00 00
```

The comments will not affect your raw exploit string. Be sure to leave space around both the starting and ending comment strings (" **`/* `**" and " **`*/ `**") so that they are properly ignored.

Below are three examples of the difference between regular string representation, our hex-formatted string, and a raw exploit string generated by **`hex2raw`**. Recall the following:

1. The ASCII representation is 30 for ‘0’, 31 for ‘1’, etc.
2. Endianness! (see Important Points / Reminders below for clarification)
3. Our exploit strings will contain byte values that do not correspond to any particular character encoding (ASCII, Unicode, etc). Nonetheless, our text editors will attempt to display our raw exploit strings to us in a readable way. Opening our exploit-raw.txt files inside of nano, vim, or another text editor will result in the weird symbols you see below.

String representation (typed into terminal)	Hex-formatted string written by you (inside exploit.txt)	Raw exploit string created by **`hex2raw`** (inside exploit-raw.txt)
“012345”	**`30 31 32 33 34 35`**	**`012345`**
`0x00012345` (address)	**`45 23 01 00`**	**`E#^A^@`**
`0x5d968887` (cookie)	**`87 88 96 5d`**	**`��h�]`**

Finally, you can use **`hex2raw`** to provide the raw exploit string to **`buffer`** in two different ways (use whichever you are more comfortable with):

1. You can set up a series of pipes to pass the string through **`hex2raw`**.

```
cat exploit.txt | ./hex2raw | ./buffer
```

The use of pipes here means that the output of **`cat exploit.txt`** will be the input of the program **`hex2raw`** and the output of running that will be the input to running the program **`buffer`**.

2. You can store the raw string in a file and then use I/O redirection to supply the file with the raw string to **`buffer`**:

```
./hex2raw < exploit.txt > exploit-raw.txt
./buffer < exploit-raw.txt
```

This approach must be used when running **`buffer`** from within **gdb**. The file containing the raw string must first be created outside of **gdb**. Then, within **gdb**, you may run:

```
(gdb) run < exploit-raw.txt
```

Important Points / Reminders

- **`hex2raw`** expects two-digit hex values separated by a whitespace. So if you want to create a byte with a hex value of 0, you need to specify **`00`**.

- The CS department machines, as well as your virtual machines, are *little-endian*, which means that the least-significant byte of a word is read first. This is why you should enter addresses into your hex string in reverse byte-order.
- Your exploit string must not contain byte value **`0x0A`** (**`0A`**) at any intermediate position, since this is the ASCII code for newline (‘**`\n`**’). When **`Gets()`** encounters this byte, it will assume you intended to terminate the string
- Your exploit strings should not cause segmentation faults. The following output of the buffer program is not valid:

  ```
  Cookie: 0x12345678
  Type string: <string>
  BEEP BEEP!: getbuf returned 0x12345678
  VALID
  NICE JOB!
  Ouch!: You caused a segmentation fault!
  Better luck next time
  ```

  More generally, it is not sufficient to receive the VALID NICE JOB! confirmation. If your goal is to get the program to set or return a particular value, it must actually do this!
- A bus error is raised when your program tries to access an invalid memory address - it’s similar to a segfault and means that your exploit did not do what you expected it to.

4 Phases

This project consists of four phases of buffer overflow attacks. The manner of attack will be slightly different in each phase.

4.1 Level 1: Lights Off (24 pts)

Our survivor camp is running low on supplies and we need a team of raiders to venture out to gather supplies! Unfortunately, the warehouse full of supplies is surrounded by Walking Doeppners! But you know that the Doeppners can only see when the lights are on outside the warehouse, so you must turn the lights off in order to get into the warehouse!

The function **`getbuf()`** is called within buffer by a function **`test_exploit()`** having the following C code:

```
void test_exploit() {
```

```
2|      int val;
3|        /* Put canary on stack to detect possible corruption */
4|        volatile int local = uniqueval();

5|        val = getbuf();

6|        /* Check for corrupted stack */
7|        if (local != uniqueval()) {
8|            printf("Sabotaged!: the stack has been corrupted\n");
9|        }

10|       if (val == cookie) {
11|            printf("BEEP BEEP!: getbuf returned 0x%x\n", val);
12|            validate(4);
13|        } else {
14|            printf("Oops: getbuf returned 0x%x\n", val);
15|        }
16| }
```

When **getbuf()** executes its return statement (line 6 of **getbuf()**), the program ordinarily resumes execution within the function **test_exploit()** (at line 7 of that function). We want to change this behavior.

```
1| /* Buffer size for getbuf */
2| #define NORMAL_BUFFER_SIZE 64

3| int getbuf() {
4|    char buf[NORMAL_BUFFER_SIZE];
5|    Gets(buf);
6|    return 1;
7| }
```

Within the file **buffer**, there is a function **lights_off()** having the following C code:

```
1| void lights_off() {
2|    printf("\"Nice!\": The lights are off!\n");
3|    validate(1);
4|    exit(0);
5| }
```

Your task is to get the buffer to execute the code for **lights_off()** when **getbuf()** executes its return statement, rather than returning to **test_exploit()**. Note that your exploit string may also corrupt parts of the stack not directly related to this stage, but this will not cause a problem, since **lights_off()** causes the program to exit directly.

Some advice:

- Writing assembly code is *not* required for this level.

- All the information you need to devise your exploit string for this level can be determined by examining a disassembled version of **`buffer`**. The command **`objdump -d buffer > obj.txt`** will disassemble the contents of the buffer executable to **`obj.txt`**. This file will then contain each function's name, with all of its instructions and the addresses of those instructions.
- Be careful about byte ordering.
- You might want to use **`gdb`** to step the program through the last few instructions of **`getbuf()`** to make sure your exploit is doing what you expect it to!
- The placement of buf within the stack frame for **`getbuf()`** depends on which version of gcc was used to compile the buffer, so you will have to read some assembly to figure out its true location.
- The space allocated for the buffer on the stack frame can be greater than the declared size of the buffer.
- It is not acceptable to jump directly to the **`lights_off()`** function call with a **`jmp`** assembly instruction. The goal of this assignment is to understand the x86-64 program stack by manipulating and exploiting it - jumping directly to **`lights_off()`** circumvents the need for any such understanding, and you will not receive credit for solutions that do this. This applies for all the phases.

Put the hex-formatted exploit string for this level in a file named **`lights.txt`** (with any comments not in your README).

4.2 Level 2: Searching for Sandwiches (24 pts)

Now that we've successfully gotten past the Walking Doeppners and into the warehouse, it's time to collect supplies! The survivor camp is full of hungry people who all love one thing: sandwiches! Since this is the apocalypse and food is scarce, the survivors are all sandwich agnostic: all sandwiches quench hunger equally. At any given time there is no guarantee that there are enough sandwiches left in the warehouse. Help the raiders find more food by making sure the warehouse's sandwich order contains enough sandwiches! The warehouse has 4 types of sandwiches and the **`sammich_types`** array describes the number of each type in the order.

Within the file **`buffer`** there is a function **`sandwich_order()`** having the following C code:

```
struct sandwich_order {
    int id;
    int sammich_types[4];
};

void sandwich_order(struct sandwich_order order) {
    int total_sammiches = find_total(order.sammich_types);
    if (order.id == cookie && total_sammiches >= 20) {
      printf("SUCCESS! You found enough sandwiches: ({0x%x, %d})  \n",
            order.id, total_sammiches);
      validate(2);
    } else {
       printf("Not enough sandwiches found! Please try again: Found ({0x%x,
          %d})\n", order.id, total_sammiches);
    }
    exit(0);
}
```

`find_total` is a function that computes the total number of sandwiches. Similar to Level 1, your task is to get the buffer to execute the code for **`sandwich_order()`** rather than returning to **`test_exploit()`**. In this example, though, you must make it look like your data is the argument to **`sandwich_order()`**. How might this work?

Some advice:

- **IMPORTANT:** You will need to set the number of total sandwiches. The number of sandwiches printed in the "Success! ..." message should be
  - Consistent between runs.
  - Equal to the arguments you provide.
- The program should not explicitly *call* **`sandwich_order()`**—it will simply execute its code. This has important implications for where on the stack you want to place your cookie and other data.
- Struct arguments are passed on the stack.
- Just like in Level 1, writing assembly is not required for this level.
- Just like in Level 1, solutions that jump directly to the **`validate()`** function call, in this case in **`sandwich_order()`**, will receive no credit.

Put the hex-formatted exploit string for this level in a file named **`sandwiches.txt`** (with any comments not in your README).

4.3 Level 3: Escaping the Warehouse (32 pts)

As you are gathering supplies, the warehouse security system goes off. The owner of the warehouse, Divyam, is following your location via trackers in the sandwiches you stole and you need to get out fast! To escape, you need to obscure your location from Divyam. Your hacker friend planted the **test_exploit** function in the warehouse's trespasser tracking system. If you could return your cookie to **test_exploit** function, the tracking system won't be able to follow your location! However, the system is sensitive to any change in the stack and will notice if anything goes wrong. Your program must not crash or have a premature exit.

Our preceding attack caused the program to jump to the code for some other function, which then caused the program to exit. As a result, it was acceptable to use exploit strings that corrupt the stack. The most sophisticated form of buffer overflow attack causes the program to execute some exploit code that changes the program's register/memory state, but makes the program return to the original calling function (**`test_exploit()`** in this case). The calling function is oblivious to the attack. This style of attack is tricky, though, since you must:

1) get machine code onto the stack,

2) set the return pointer to the start of this code, and

3) undo any corruptions made to the stack state.

Your job for this level is to supply an exploit string that will cause **`getbuf()`** to return your cookie back to **`test_exploit()`**, rather than the value 1.

You can see in the code for **`test_exploit()`** that this will cause the program to go "**`BEEP BEEP!`**". Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a ret instruction to really return to **`test_exploit()`**.

Some advice:

- **IMPORTANT:** in your machine code, you want to be returning to previous execution. You must
  - return to the exact position of where execution left off (you cannot skip lines in `test_exploit`).
  - not use the `jmp` or `call` instructions.
- You can use gdb to get the information you need to construct your exploit string. Set a breakpoint within **`getbuf()`** and run to this breakpoint. Determine parameters such as the saved return address.
- There is no need to determine the byte encoding of assembly instruction sequences by hand. Instead, you can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with gcc and disassemble it with **`objdump`**. You should be able to get the exact byte sequence that you will type in your hex-formatted string. See the section Generating Machine Code for a walkthrough of how to do this.
- Keep in mind that your exploit string depends on your machine, your compiler, *and* your generated cookie.

Put the hex-formatted exploit string for this level in a file named **`lost.txt`** (with any comments not in your **`README.md`**).

4.4 Level 4: Fraudster (20 points)

After successfully making it out of the warehouse and past the horde of Walking Doeppners surrounding it, you find that an even larger horde is heading your way! Bypass the new stack security measures so you can make it back to the camp unscathed!

For this phase you'll need to run the **buffer** program in "nitro mode" by using the **-n** command-line flag.

From one run to another, especially by different users, the exact stack positions used by a given procedure will vary. One reason for this variation is that the values of all environment variables are placed near the base of the stack when a program starts executing. Environment variables are stored as strings, requiring different amounts of storage depending on their values. Thus, the stack space allocated for a given user depends on the settings of his or her environment variables. Stack positions also differ when running a program under **gdb**, since **gdb** uses stack space for some of its own state.

In the code that calls **getbuf()**, we have incorporated features that stabilize the stack, so that the position of **getbuf()**'s stack frame will be consistent between runs. This made it possible for you to write an exploit string knowing the exact starting address of **buf**. If you tried to use such an exploit on a normal program, you would find that it works sometimes, but it causes segmentation faults at other times.

For this level, we have gone the opposite direction, making the stack positions even less stable than they normally are.

While the location of the return address on the stack changes each time the program is run, the actual address of the code to return to doesn't change, since it's only the stack that's at a different location -- the location of the code itself doesn't change. Remember that the stack and actual program code live in very different regions of virtual memory! (The code for any function, like test_exploit and getbuf, does not live on the stack -- only their stack frames do.)

When you run **buffer** with the command line flag "-n," it will run in "Nitro" mode. The program calls a slightly different function, **test_exploitn()**, which calls **getbufn()**:

```
int getbufn() {
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

This function **getbufn()** is similar to **getbuf()**, except that it has allocated enough space in the buffer for 528 characters. You will need this additional space to create a reliable exploit. The code that calls **test_exploitn()** (which calls **getbufn()**) first allocates a random amount of storage on the stack, such that if you sample the value of **%rsp** during any two executions of **test_exploitn()** or **getbufn()**, you would find they differ by as much as 240. As a result, the addresses you used to solve previous phases may not work in this phase.

In addition, when run in Nitro mode, **buffer** requires you to supply your string 5 times, and it will execute **test_exploitn()** and **getbufn()** 5 times, each with a different stack offset. Your exploit string must make it return your cookie each of these times.

Conceptually, just because one of the five runs works doesn't mean that it always will. Since the address of the buffer is changing between runs, it's possible that the one address that you've identified only works for that particular run.

Your task is almost identical to the task for the previous level. Once again, your job for this level is to supply an exploit string that will cause **`getbufn()`** to return your cookie back to **`test_exploitn()`** rather than the value 1. This will cause the program to go "**`Woo`**." Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a ret instruction to really return to **`test_exploitn()`**.

Some Advice:

- **IMPORTANT:** in your machine code, you want to be returning to previous execution. You must
    - return to the exact position of where execution left off (you cannot skip lines in **`test_exploitn`**).
    - not use the `jmp` or `call` instructions.
- Recall that the space allocated for the buffer on the stack frame can be greater than the declared size of the buffer.
- Make sure to run with the **`-n`** flag!
- If you want to use GDB with the **`-n`** flag, do "gdb buffer". After you've set breakpoints, you can then do "run -n" so as to pass the -n flag to buffer.
- You can use the program **`hex2raw`** to send multiple copies of your exploit string by providing it with the command-line argument **`-n`**. Using **`hex2raw`** with the command-line argument **`-n`** will send 5 copies of the exploit string provided. The command-line argument **`-n`** for buffer will execute the buffer program in "Nitro" mode which expects 5 exploit strings. If you have a single copy in the file **`fraudster.txt`**, then you can use the following command:

```
cat fraudster.txt | ./hex2raw -n | ./buffer -n
```

- You must use the same string for all 5 executions of **`getbufn()`**.
- The trick for this phase is to make use of the nop instruction. It is encoded with a single byte (code **`0x90`**). By including a sequence of nop instructions before your exploit code, if the program jumps to any point in the sequence, it will "slide" along until it reaches the exploit code. Such a sequence of nops is known as a *nop sled*. More information about nop sleds can be found in the lecture slides.

Put the hex-formatted exploit string for this level in a file named **`fraudster.txt`** (with any comments not in your README).

5 Generating Machine Code

Your assembly code (exploit) should be converted into hex values and then put into a text file to run level 3 and level 4.

Using **`gcc`** as an assembler and objdump as a disassembler makes it convenient to generate the bytes for instruction sequences needed for our raw strings. For example, suppose we write a file *example.s* containing the following assembly code:

```
# Example of hand-generated assembly code
      push $0xabcdef # Push value onto stack
      add $17,%eax # Add 17 to %eax
```

Anything to the right of a '**`#`**' character is a comment.

We can now assemble and disassemble this file with the following commands:

```
gcc -c example.s
objdump -d example.o > example.d
```

The generated file example.d contains the following lines:

```
0:    68 ef cd ab 00          push   $0xabcdef
5:    83 c0 11                add   $0x11,%eax
```

Each line shows a single instruction. The number on the left indicates the starting address (starting with 0), while the hex digits after the ':'. character indicate the byte codes for the instruction. Thus, we can see that the instruction **`push $0xABCDEF`** has hex-formatted byte code **`68 ef cd ab 00`**.

Finally, we can read off the byte sequence for our code as:

```
68 ef cd ab 00
83 c0 11
```

These characters can then be copy and pasted, as is, into your exploit.txt files. Remember that you will still have to call **`hex2raw`** on these files to create a raw exploit string!

6 gdb

Here are some gdb commands that you may find helpful for this assignment:

- **`x/i $pc`** prints the current instruction. **`pc`** stands for program counter, which is an alternative name for instruction pointer.
- **`disassemble <function>`** prints each instruction (and its address) of <function>.
- **`p <function>`** prints the address of <function>.

- **`i r`** prints the value contained in each register.
- **`i r rsp`** prints the value contained in the rsp register, specifically.
- **`x/48b`** prints 48 bytes of memory. **`x/48b <address>`** prints the 48 bytes of memory after <address>, and **x/48xb** will print 48 bytes of memory in hex.
- **`si`** steps over a single x86-64 instruction. **`step`** and **`next`** won't do this sometimes, but **`si`** never fails to do so. Use this command well.
- **`ni`** is like **`si`**, but will execute function calls as single instructions (it will not enter the function call like **`si`** will).
- **`layout asm`** will display the assembly instructions.
- For more gdb commands, consult the GDB Cheatsheet.

Set breakpoints frequently and use these commands if you get stuck.

However, do not set a breakpoint on instructions which you have placed on the stack. Doing so may cause a null byte, **`0x00`**, to replace one of your instruction bytes, which will very likely ruin that instruction (and the ones following it).

7 README

You should have a **very detailed** explanation of each of your exploits in **one of two places**. Your first option is to write a **`README.md`** file (which is required to hand in anyway) in which you explain how each of your exploits works (i.e. each component of your exploit). Your second option is to use block comments (i.e. **`/*...*/`**) within each of your exploit text files to explain the exploit, and then hand in a **`README.md`** saying that your explanation of your solutions are in the text files.

Every block of your exploit string should be commented. You should explain the purpose and reasoning behind each of the blocks. If in doubt, more is always better than less! If there is anything unclear about commenting please go to TA hours, or ask on Ed. Comments are a significant portion of your grade.

8 Grading

You will receive points for successfully exploiting each level and for your explanation.

- Level 1: Turning off the lights and the explanation are worth **24 points** (explanation is worth **4 points**).
- Level 2: Stealing sandwiches for your camp and the explanation are worth **24 points** (explanation is worth **4 points**).
- Level 3: Escaping the warehouse and the explanation are worth **32 points** (explanation is worth **6 points**).

- Level 4: Becoming a fraudster and the explanation are worth **20 points** (explanation is worth **4 points**).

The autograder will only give you points for passing exploits, thus giving a score out of 82. This assignment is worth 100 points total. The explanations will be graded based on what is in your **`README.md`** (if you choose to use block comments within your exploit files, your **`README.md`** must mention that explicitly). You will be penalized if you do not explain your work.

See the table below for guaranteed grade cutoffs. If you do not meet the threshold for a given letter grade, you may still receive that grade after Professor Doeppner releases final cutoffs for the project (you will only ever be curved up).

Grade	**Requirements**
A	Level 4 passes the autograder, with explanations
B	Level 3 passes the autograder, with explanations
C	Levels 1 + Levels 2 pass the autograder, with explanations
Failing	Does not pass any of the autograder tests for any levels

9 Handing In

Your handin for this assignment should include at least a text file for each phase containing your input string for that phase, plus your cookie.txt file and a **`README.md`**:

- `cookie.txt`
- `lights.txt`
- `sandwiches.txt`
- `lost.txt`
- `fraudster.txt`
- `README.md`

That is, the commands

```
cat <phase>.txt | ./hex2raw | ./buffer
```

and

```
cat fraudster.txt | ./hex2raw -n | ./buffer  -n
```

should solve the indicated phase (the autograder runs these commands). There is no need to include the stencil files, but it won't affect the autograder if there are extra files submitted, as long as you submit the six files given above!

If this is your first time handing in, or you need a refresher, please refer to the Github & Gradescope guide. This will walk you through how to upload your work to Gradescope using your Git repository created in Installation.

If the autograder does not seem to reflect your local changes, be sure to have the correct branch selected, and push your latest changes to it!