9

Lab 7: Stack Smashing, Part 2

In this assignment, you will continue constructing a stack-based buffer overflow attack. Here, we carry out the primary aim of the attack: extracting a secret value. To do this successfully, you will need to write your own attack code in assembly, tying together existing functions to exfiltrate a value without entering in a password. Although it is not strictly required for this attack, we will also explore techniques to make your attacks work against a broader set of C string-handling functions.

For each question, be sure to follow the instructions carefully, supplying all of the parts mentioned. You are strongly encouraged to supply a Makefile that produces whatever artifacts you submit. Please make sure that your Makefile includes updated `all` and `clean` targets.

9.1 Learning Goals

In this lab, you will learn:

- How to use the analysis skills from Lab 5 to plan a novel attack.
- How to write that attack in ARM assembly.
- How to make your attack robust to string-handling functions.

9.2 Requirements

Language. In order to carry out the attack you will primarily write assembly code. You may also need to write small utilities in C in order to prepare your attack. Hand in all of the utility programs you write along the way.

Common environment. Your code must be developed for and work on the Raspberry Pi machines we use for class.

Stack Overflow and the honor code. You are permitted to refer to Stack Overflow for help, but you must not under any circumstances copy the code you see there. If you find a helpful Stack Overflow post, you *must attribute the source of your inspiration in a comment at the appropriate location of your code.* You must also provide the URL of the post. Unattributed code will be considered an honor code violation.

Reflection questions. This assignment asks you to answer a few questions. You must supply the answers to these questions in a `PROBLEMS.md` file.

Starter code. For this assignment, your repository includes the program you need to exploit and a `Makefile`.

9.3 Lab 5

This lab builds on the skills you learned while working on Lab 5. If you want to revisit your Lab 5 solution, you are welcome to do so. Simply edit your code and push—no need to tell me. I will grade Labs 5 and 7 all together.

9.4 Step 1: Jump to a function that takes input

Your first attack should call the `test` function, which returns a `char *`. Pass this returned `char *` to the `test3` function, which prints it out.

You can find the address of an arbitrary function in GDB using the `disas` function. For example, `(gdb) disas test` will jump the `gdbtui` display to the location of the `test` function.

Note that since you are doing something more sophisticated than simply jumping to an arbitrary address, you will need to utilize an argument as in the previous attack. Again, you will exploit this program by crafting an input.

This input will likely rely on custom shellcode, written by you. There are two approaches to writing shellcode:

- Write a C program and generate assembly to use as inspiration.
- Hand-craft assembly.

In both cases, you will likely need to refine your shellcode by hand. Aside from choosing instructions carefully, there are some techniques that make life much easier:

- Move the stack base to a safe location so that it does not interfere with your carefully-crafted shellcode. Locations at a lower address than the target buffer are probably safe (i.e., "above" the stack). Remember that that functions you call *expect that the C call stack exists and functions correctly*.
- Use the `.asciz` assembler directive to insert a string literal directly into code. Then use the `adr` instruction to load the address of the label into an instruction. For example:

```
  adr r0, thing
  ...
thing:
  .asciz "hello world!"
```

- Because all ARM instructions are exactly 32 bits wide, this makes utilizing full 32-bit numbers cumbersome. Most ARM instructions can only accommodate 8-bit immediate values. Here are some workarounds:
  - Use the `.word` assembler directive with `adr` and `ldr` to put the value into a register. For example:

    ```
      adr r0, a_number
      ldr r1, [r0]
      ...
    a_number:
      .word 12345678
    ```

  - Use addition and bit-shifting to create a number. For example, to obtain `0xabce` from `0xab` and `0xcd`, you can do:

    ```
    mov r0, 0xab
    lsl r1, r0, #2
    mov r0, 0xcd
    orr r0, r0, r1
    ```

  - The `ror` instruction is a special `mov` instruction that lets you move and rotate an instruction all in one step. See the ARM KEIL manual for details.

- The bl instruction cannot jump to an address stored in a register, which is inconvenient; it only works with immediates. Fortunately, blx can take a register operand, and like bl, it also saves the return address in the lr register.

  Be sure to supply:

1. your input as a string of escaped hexadecimal literals in a file called input3.hex;
2. your input in binary in a file called input3;
3. your shellcode as an assembly program called input3.s; and
4. an explanation how your attack works in the PROBLEMS.md file.

You are encouraged, but not required, to supply a Makefile that builds the above artifacts. Specifically, consider having targets for input3.o, input3.hex, and input3 using a Makefile. Doing so keeps your code organized and it makes it easy for me to follow your train of thought.

Note: Be sure to put your work in the part2 folder.

9.5 *Step 2: Remove NULL bytes from input3*

Although removing NULL bytes is not strictly required to make this attack work, for full credit, you will need to ensure you have removed NULLs from your input. Removing NULL bytes ensures that if your attack input is subsequently handled by a C string function that checks for the presence of NULL bytes that it passes through those functions in its entirety.

You can check for the presence of NULL bytes in your attack binary using the objump and hexdump tools. NULL bytes in assembled code comes from two different sources:

1. *Instructions themselves.* For example, the program eor.s

```
main:
  eor r0, r0
```

   produces NULL bytes:

```
$ objdump -d eor.o

eor.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
   0: e0200000        eor     r0, r0, r0
```

Observe that this instruction is encoded on disk as `00 00 20 e0`.

2. *Literal values*. For example, the word `0xff` is actually represented on disk as the little-endian word `ff 00 00 00`.

Aleph One's paper, "Smashing the Stack for Fun and Profit," gives some background on `NULL`-removal. There are many approaches to removing them. In general, these approaches call for some creativity. Try to think of this problem as a fun puzzle.

- `eor` a register with itself to obtain zero values.
- Assemble values using multiple instructions. For example, logical shifting to set high or low bits.
- Storing "proximate" numbers using `.word`, which you then modify at runtime (e.g., using shifts, addition, etc). For example, the byte `0x01` is "close" to the byte `0x00`.
- `.asciz` is handy precisely because it automatically `NULL`-terminates strings for you. Unfortunately that runs counter to our goal of `NULL` byte removal. Instead, use `.ascii`, which does not `NULL`-terminate. If you plan to give a string created by `.ascii` to a C function, remember that it *must* be `NULL`-terminated. You will have to `NULL`-terminate it at runtime.

The `shellcode-test.s` and `shellcode.s` programs distributed with Lab 5 utilize all of these tricks. See that code for examples.

Be sure to supply:

1. your input as a string of escaped hexadecimal literals in a file called `input4.hex`;
2. your input in binary in a file called `input4`;
3. your shellcode as an assembly program called `input4.s`; and
4. an explanation how your attack works in the `PROBLEMS.md` file.

As before, you are encouraged, but not required, to supply a `Makefile` that builds the above artifacts.

9.6 *Step 3: Call the `decrypt` function*

Your second attack should call the `decrypt` function with an arbitrary input that is *not* a valid student ID. This attack will be similar to the previous attack in that you will need to utilize an argument in order to feed an input to the `decrypt` function. Observe[1] that `decrypt` returns a `char *`. To print it, you will need to call some kind of print function, like in Step 1.

[1] By looking in `enc.h`.

Again, you will exploit this program by crafting an input. Once exploited, you will trigger a fault handler[2] in the program that will return a pointer to one of a set of strings.

[2] The handler prints an error message backward. The purpose of this handler is to let you know when you're on the right track.

Your attack code should be supplied with `NULL` bytes removed. However, you are encouraged to start with an ordinary assembly program containing `NULL` bytes if you are struggling with that step.

1. your input as a string of escaped hexadecimal literals in a file called `input5.hex`;
2. your input in binary in a file called `input5`;
3. your shellcode as an assembly program called `input5.s`;
4. supply one of the outputs of the above code in your `PROBLEMS.md` file; and finally
5. provide an explanation how your attack works in the `PROBLEMS.md` file.

As before, you are encouraged, but not required, to supply a `Makefile` that builds the above artifacts.

9.7 *Step 4: Call the `decrypt` function with your student ID*

Your final attack should call the `decrypt` function with your own Williams ID. The program will return a value that is unique to your ID.

Your attack code should be supplied with `NULL` bytes removed. However, you are encouraged to start with an ordinary assembly program containing `NULL` bytes if you are struggling with that step.

1. your input as a string of escaped hexadecimal literals in a file called `input6.hex`;
2. your input in binary in a file called `input6`;
3. your shellcode as an assembly program called `input6.s`;
4. supply the URL given in the output of your code in your `PROBLEMS.md` file; and finally
5. an explanation how your attack works in the `PROBLEMS.md` file.

As before, you are encouraged, but not required, to supply a `Makefile` that builds the above artifacts.

9.8 *Submitting Your Lab*

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab05-07_stack_smashing-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

9.9 *Bonus: Extra Challenges*

There are two possible extra challenges.

1. The first is to carry out this attack in such a way that it does not produce a segmentation fault. Doing so will require that you think carefully about how the attack should modify (and possibly preserve) parts of the stack.

2. The second bonus possibility is to decrypt all of the stored values.

   In either case, be sure to explain how your attack works in the `PROBLEMS.md`` file.

9.10 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super easy, ..., 5 = super hard)? Why?
2. Did this assignment help you to understand buffer overflow attacks?
3. Is there is one skill/technique that you struggled to develop during this lab?
4. Your name, for the bonus point (if you want it).

9.11 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass the auxiliary sensor" you should have
    written "bypass the primary sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too many to count" as you
     state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.