CSE 509 Programming Assignment 1

September 10, 2024

1 Overview

It is recommended that this project be done by pairs of students. You can, of course, choose to do it individually, but it is obviously going to be more work. Besides, exploit writing is an inexact process, so there may be times when you get stuck. With two people working on the assignment, it is less likely that both will get stuck in the same way; and even if you do, you can work in parallel to find a work-around.

The grading criteria will be different for individuals and pairs: for individuals, fewer parts will be mandatory, while the remaining parts will fetch extra credits. Extra credit for submissions can be up to 40%, i.e., you can score up to 140 points out of 100, but finishing all the parts will take substantially more time than completing the mandatory parts.

I anticipate that this will be the longest of all programming assignments, and so it will carry more points than the other assignment. Plan on starting the work on this assignment right away. *It would be more appropriate to call this a "debugging assignment" rather than a programming assignment. The amount of coding involved is rather small.*

You are given a vulnerable program `vuln.c`. This program, together with a Makefile, is provided as a tar-gzipped archive. Note that `vuln` accepts commands on its input and executes them. Examine the source code to see what the commands are. (Until you read that code, you cannot fully understand the rest of this assignment description.)

You are permitted to discuss the problem on Piazza, but don't go to the level of posting your code. You can post a small pseudo-code snippet that you are trying to understand or have difficulty with, and others can clarify/explain. *Unless you do the assignment yourself, you will have a hard time solving some of the problems in your exams.* Also note that memory layouts are different for each group/individual, so the exploit that works for one group will fail for another group. This variation is implemented using the environment variable `GRP_ID` that should contain your group's group id. (By default, it will be the student ID number of the team member whose last name is alphabetically the first among the group members. Course staff will contact you if there is a difference from this default.) Since the exploits are different with different groups, I can make a fully working sample exploit for the data-only exploit. This exploit works when you set `GRP_ID` to 1000. This example will give you a road map on how to construct your exploit code, and how to structure it.

Note that `vuln` uses `read` rather than `scanf` or `gets`. This means you can input arbitrary values as input, a capability you need if you want to input arbitrary binary data that may include code or pointer values.

There are three basic vulnerabilities that you can exploit:

- a format string vulnerability in `main_loop`,
- a heap overflow vulnerability in the version of `malloc` defined in `my_malloc.c` and used in `vuln.c`,
- a stack overflow vulnerability in `auth`.

Some of these vulnerabilities can be exploited in more than one way.

Note that you don't need to disable ASLR, stack protection or fool around with $W \oplus X$ to get your exploits to work. Instead, you will use the printf and buffer overflow vulnerability to leak as much of the memory contents as you want. Initially, you will leak the contents of the stack. The stack will contain stack cookie — gcc uses the same value of the cookie for all functions, so you can read and reuse them. The stack will also contain saved base pointer. By reading it, you can overcome randomization of the stack base address. To cope with randomization of code memory, you can read the return addresses off the stack. By dumping code memory, you can read information such as the address of functions in libraries (e.g., `bcopy`), and from there, you can compute the location of a more useful function such as `execl`. Finally, to overcome $W \oplus X$, note that the Makefile already makes the stack executable.

Note that Makefile automatically disassembles the vulnerable executable `vuln` to produce `vuln.dis`. To make the assembly file easier to understand, it embeds source code lines within assembly, so that you will know what line of source code results in which assembly instructions. The disassembly is produced by the `objdump` tool. (See the `Makefile` for details.)

2 Exploits

2.1 Data-only attack

Use a data-only-attack on the local variable `db` in the `auth` function. In particular, use stack smashing in `auth` to overwrite `db` so that it points to the location of another local variable `cred` of `auth`. This causes the `chkPw` function to be called with two identical arguments, in which case it will always return true. By subverting the checking logic this way, an attacker can login without providing a valid username or password.

- This attack does not corrupt any critical data on the stack: no cookies or return addresses are affected.
- *You are given a working version of this exploit for* `GRP_ID` *1000. You need to modify it so that it works for your group's id.*
- Make sure that your attack does not corrupt stack cookies, base pointers, return addresses, or anything else. Otherwise you will not get full credit for this exploit.

2.2 Return-to-helper2

Use a return-to-libc attack that returns to `private_helper2`. Since `vuln` is compiled into a position-independent executable (PIE), the location of `private_helper2` will be different each time you run it. So, you will need to leak some code address (e.g., a return address) using the format string vulnerability, and use it to compute `private_helper2`'s location. You will also need to leak the stack canary using the printf vulnerability.

If you end up completing Return-to-helper, then don't submit Return-to-helper2. You will be given credit for both parts if you just submit a working exploit for Return-to-helper.

2.3 Return-to-helper

Return to `private_helper`, while controlling its arguments. Specifically, the exploit should result in the printing of

```
**** private_helper(0x12345670, 0x123456789abcdef0, <some_pointer> "/bin/sh") called
```

On x86_32, controlling the arguments will be easy because they are all on the stack. Since your exploit overwrites the return address, it would be simple to extend it to overwrite the next several bytes that will be interpreted as parameter values. Unfortunately, parameters are passed through registers on x86_64, so this simple option is not applicable. However, with some work, poring over the assembly code of `private_helper` in `vuln.dis`, you can find a work-around. Compilers frequently save registers on the stack so that these registers can be used again, e.g., when the callee in turn wants to make another call, and needs to set argument registers for this call. When the old register values are needed, they are loaded from the stack. See how you can exploit this fact to control the arguments of `private_helper`. Keep in mind that in assembly code, you can jump to or call any instruction, and are not necessarily limited to the first instruction of functions.

To do this exploit successfully, you will also need to think carefully about the contents of `rbp`. If necessary, control the value of the saved ebp on the stack as part of your exploit.

2.4 Return-to-injected code

Implement an exploit that returns to injected code on the stack. Your injected code, in turn, should call `private_helper()`. Note that since parameters are passed in registers, it is easy to control them in your injected code: include instructions to load the desired values into the parameter registers. Finally, jump into `private_helper`.

To get credit, `vuln` should print the exact same message shown above for the return-to-helper attack. Keep in mind that it is not that convenient to use a direct jump to `private_helper`, as the operand of this instruction is the distance between the source and target. Since the source instruction is on the stack and the target in the text segment, such a jump may not be possible. So, it is better to use an indirect jump, which involves loading the target into a register, and jumping using that register.

Note that in some instances, you may not know the exact starting address of injected code. In those cases, attackers precede their code with a *NOP-sled*. This is simply a sequence of NOPs, which are 1-byte instructions in the x86 architecture. Now, you can jump into any byte of the NOP-sled, and then execution will flow through the NOPs to the following code.

2.5 Partial Overflow Attack

This attack does not require the format string vulnerability, so you won't receive credit if you rely on information leaks. In fact, you should not rely on any vulnerability other than the stack-smashing vulnerability in `auth`.

The goal of this attack is to receive the "service" that `vuln` provides without having to provide a valid username/password, *and* without using the data-only attack. One way to achieve this is by returning from `auth` to a different location in `g` than the one from which `auth` was called.

To carry out this attack without any information leaks, you are going to rely on the endianness of x86_64. In particular, you can overwrite the least significant bytes of a multi-byte integer *without overwriting the remaining bytes.* Using this feature, you will first try to overwrite the LSB of the canary. If you get it right, `vuln` will report a failed login. If you got it wrong, it will be terminated. Since `vuln` is a fork-based server, the parent `vuln` process will immediately fork another child, which will then print a "Welcome" message. You can read this message in the driver program, and determine if you guessed the canary right.

As discussed in the class, you can cycle through all 256 possible values of each byte of the canary, so you can guess the canary correctly in about a thousand or so guesses. It took my program about one second to cycle through this many guesses. If it takes significantly longer for you, then you are probably doing something wrong.

Important Note: I found that `apport`, a service that processes crash reports (and possibly sends them to a bug repository) can slow down the above process by more than 100 times! So, ***make sure that you disable apport in your VM using the command*** `sudo service apport stop`.

Once you have got the canary, then you are going to attempt a partial overwrite on the return address. Specifically, you can overwrite the LSB of the return address, and this is enough to send the return to anywhere within 256-bytes of the original return address. Moreover, the way your executable is randomized, it preserves the last 12-bits of the locations of every instruction, making it easy to figure out the offset to jump to. If you target the right location, `vuln` will indicate to you that your authentication succeeded, and it is providing you the requested service (which, we have mocked with the `ls` program).

2.6 Format String Attack

Implement an attack that uses only the format string vulnerability. Your goal is to execute arbitrary code injected by the attacker. Your injected code can simply call `private_helper2`. This is an extra-credit problem for teams as well as individuals.

For this attack, *you should not overwrite the canary* — you should selectively target the return address of `main_loop`, so that execution is diverted to the injected code when the quit command is sent to `vuln`, and it returns from `main_loop`.

You can't begin on this attack until you carefully review and fully understand the file `formatstr_notes` ***in the*** `pub/` ***directory in the tarball you are given.*** That document provides very detailed instructions for structuring your attack on a 32-bit system. It should be an easy extension to work with 64-bit.

2.7 Heap Overflow: Parts A and B

Note that the heap overflow vulnerability resides within `heap_delete` function in `my_malloc.c`. This function is called from `my_malloc` as well as `my_free`. In theory, one could exploit it from either place. However, the heap blocks have to be arranged in a certain way in order for this work. So, you may need to use the `u`, `p` and `l` commands a few times to make sure that heap blocks are ordered in just the right way for your attack to work.

- **A: Heap metadata overwrite in** `my_free` (*Extra credit*): Exercise a heap overflow in `my_free` to overwrite return address on the stack so that when `main_loop` returns, it executes `private_helper2`.
- **B: Heap metadata overwrite in** `my_malloc` (*Extra credit*) Exercise a heap overflow in `my_malloc` to overwrite return address on the stack so that when `main_loop` returns, it executes `private_helper`.

To get this to work, you need to understand the implementation of `my_malloc` to a certain extent. In particular, you need to know the size of the blocks, the order in which the blocks occur in the free list, etc. You need to know the order because you can only overflow from a block starting at a lower address to a block beginning at a higher address.

Another challenge is that in `my_free`, there are two assignments:

```
current->next->prev = current->prev
```

and

```
current->prev->next = current->next
```

You can use the first statement to assign arbitrary value (contained in `current->prev`) to an arbitrary location (contained in `current->next`). Unfortunately, the second statement will interpret `current->prev` as an address and write to it. So, `current->prev` cannot point to the code segment. This means that your payload must be in writable memory, i.e., you need to execute injected code; it is not possible to do a return-to-existing-code attack.

You also need to figure out where your exploit code is going to reside. If you expect it to be in the heap block, then, keep in mind that the base of the heap managed by `my_malloc` is randomized, and will differ across runs. So, you need to figure out how to use the printf vulnerability to extract the base address. Alternatively, you can see if the exploit code can be put on the stack, whose addresses you have already figured out. (This is what I did.)

3 Submission

Your submission will be in the form of C-programs. In particular, for each exploit, you will create a version of the driver program. Compiling and running this exploit program should lead to a successful exploit. *Note that you need to submit the source code for the exploits.* You should not change `vuln.c` or any of the other material provided to you.

You should create a tar-gzipped archive of all your exploit programs. Give them descriptive names such as `driver-ret2helper2.c`. ***Do not submit any of the code we give you. You should only submit the source code of the driver programs you wrote.*** Submission will be on Brightspace.

4 Distribution of Points

Points are shown in italics for extra-credit problems and normal face for required problems. Note that the points are different for individuals and groups, so they are shown in different columns. This assignment is graded on a 100-point scale, but the maximum possible is 140.

Problem	**Points**
Data-only	30
Return-to-helper2	15
Return-to-helper	20
Return-to-injected	35
Partial overwrite	*20*
Format string	*20*
Heap overflow A	*20*
Heap overflow B	*20*
Total	140

Table 1: Point distribution for individuals

Problem	**Points**
Data-only	23
Return-to-helper2	11
Return-to-helper	14
Return-to-injected	25
Partial overwrite	27
Format string	*20*
Heap overflow A	*20*
Heap overflow B	*20*
Total	140

Table 2: Point distribution for 2-person groups

Although extra credit points can add up to a lot, please keep in mind:

- You can get extra credit only after solving the required problems (*First four problems for individuals and first five problems for groups*). Specifically, if you are missing more than one required exploit, then you cannot get any extra credit.
- Although the points may add up to a larger sum, the total extra credit points will be capped at 40 for both individuals and groups. Individuals can solve any two of the four extra-credit problems to earn 40 points, whereas groups can solve any two out of three.
- As compared to the required exploits, the extra credit exploits will earn you far fewer points per unit time spent. You should attempt them only if are genuinely intrigued by the problem, and want to solve it even though it is going to mean a lot of effort and time.

5 Tips

- Use the 64-bit VM image provided to you. Your submission will be tested on this VM, so you might as well work on the same VM.
- **Don't change the `Makefile`**, except possibly for adding additional lines for compiling additional exploit programs.
- **Review carefully the example exploit program `driver_auth_db.c`**. You will gain a better understanding of how to structure your exploits, and also save time on other exploits.
- You can print a specific offset that is, say, 100 words from the top of the stack using `printf("%100$x")` instead of having to use 100 instances of `%x`'s. (Note that this may end up printing something that is a few words off, say, 97 words from the top of the stack.)

- Within gdb, registers can be accessed by prefixing them with `$`, e.g., `print $rsp` will print the stack pointer register.

- Within gdb, you can print arbitrary memory locations by casting them into pointers and dereferencing them, e.g., `print *(int *)0xbfffff7c`. You can control the format, e.g., print it in hex using `print /x *(int *)0xbfffff7c`.

- To print many memory locations, use the `x` command. For instance, `x /16x $rsp` will dump 16 words from memory, starting from the address in the stack pointer register. The second `x` in the command indicates that you want the values printed hex format.

- *Parameter passing convention in 64-bit Linux/x86:* The registers RDI, RSI, RDX, RCX, R8, and R9 are used for passing the first six integer-type arguments. Arguments 7+ will be passed on the stack. (Floating point arguments are passed in other registers, but you don't have to worry about floats in this assignment.)

- You need to use the printf vulnerability to leak several pieces of information. The first is the saved rbp value that you need in order to figure out the base of the stack frames. (You cannot hard-code stack base address because the stack base is (re)randomized on each execution.) The second is the return address on the stack, or the address of library functions in the GOT (Global Offset Table).

  The driver program is necessary because of the need to leak these pieces information. You will structure your exploits as follows. First, you will use the `e` command to leak the above pieces of information. You will extract the information into variables in the `driver` program, which will then construct an exploit string and send it to `vuln`.

- You can debug an already running process by using `gdb` to attach to it. (On recent Linux versions, you will need to run gdb as `root` in order to attach to an existing process.) To attach to an existing process, e.g., `vuln`, type `ps ax|grep vuln` at the bash command prompt. It will produce a list of processes that have the name `vuln`. Note down the pid, fire up gdb, and at its command line, type `attach` to that pid.

  This ability is invaluable for tracking down problems with your exploits.

- If you want to do the extra-credit problems, then first use `objdump` to disassemble the executable. *An executable contains code that won't be in the object file* `vuln.o`*, or the assembly file* `vuln.s`*.* Use `objdump -d vuln` to disassemble the executable. Then you will see how library calls are made, and how you can hijack them.

Although the stack and code layout is going to be different for each team, the layout does not change from one run to another. So you can use gdb to figure out the layout once, and then use it repeatedly in your exploits. Specifically, you need to know the size of the stack frames of `main_loop` and `auth`, and you can find this by running `vuln` within gdb, setting break points in these functions, and printing the values of `rbp` and `rsp` registers. Make sure that you print `rsp` value after the calls to `alloca`. (This function allocates storage on the stack, and hence will change the value of `rsp`.)

In order to succeed in this project, you have to get good at using gdb if you are not already there.

5.1 Working with assembly/object code

Some exploits require you to use binary code. You can do this by writing a small assembly code snippet and then compiling it using an assembler. One option is to use `as`, the default assembler on your system. You can invoke it as:

```
as -a --64 test.s
```

where `test.s` is the file containing your assembly code. This command dumps the assembled code on the screen. Note that `as` uses AT&T syntax for assembly. Alternatively, you can `nasm` which supports Intel format. (I have not used nasm.)

Instead of trying to use direct jumps or calls to absolute memory locations, you should try to use indirect jumps and indirect calls. First move the target address into a register, and then use an indirect jump or call using that register. Various other points to note:

- Make sure you get your assembly syntax right for various addressing modes and operands. Specifically, for `as`, make sure you prefix immediate operands with a `$`, and register operands with a `%`. For instance, `mov $0x20, %rax` moves the decimal number 32 into the register `rax`, while `mov 0x20, %rax` moves *the contents of memory location* `0x20` into `rax`. Also make sure that you use a `*` for indirect calls and jumps, e.g., `call *%rax` is an indirect call to the address contained in `rax`. (However, `call *(%rax)` first dereferences the location whose address is in `rax`, and then fetches the value stored at this memory location, and then calls that location.)
- You can use `gdb` to work at the assembly level. You can use `layout asm` to see your code in assembly. You can use `stepi` to single-step assembly instructions. To revert back into source code view, use the command `layout src`. Here are two helpful references (just about two pages each) on gdb and assembly code.
  - `http://web.cecs.pdx.edu/~apt/cs491/gdb.pdf`
  - `https://sourceware.org/gdb/current/onlinedocs/gdb/Machine-Code.html`