PeanutPower: CS557 Project 2

Thomas Le Baron
Team 1008
Computer Science
Worcester Polytechnic Institute
Worcester MA, USA
tlebaron@wpi.edu

Brittany Lewis
Team 1008
Computer Science
Worcester Polytechnic Institute
Worcester MA, USA
bfgradel@wpi.edu

***Abstract*—In this work we introduce a vulnerable binary ”PeanutPower” which has a format string vulnerability. We also prove that this binary can be explioted to launch bin shh through execve. We do this through overwriting the global offset table using a format string vulnerabilitiy. We are then able to use return-oriented programming to utilize a series of gadgets that allow us to launch a shell.**

I. Finding PeanutPower

Our docker image for PeanutPower is located at:
https://hub.docker.com/r/0495/cs557-peanutpower/
The challenge binary is:
home/user/penutpower
The solution is at:
/root/solutions/penutpower/
Hashes obtained using linux md5sum:
The hash for the binary is:
db9526e5b578370f2ba74cee1b4760fc
The hash for the source file is:
c9aebb24c506994dd6faeaf3d4b13622

II. Introduction

This project aims to demonstrate our knowledge on code-reuse attacks. After code injection attacks, defenses like the NX bits has been developed. In this case, it enforces the fact that code passed on the stack or the heap should never be executed, since they should just be used to store in memory. With stack and heap not executable, we can not execute injected shellcode. The idea of code-reuse attack is to execute pre-existing code (code we did not inject). The basic code-reuse attack is to execute the system() function of the libc by passing the ”/bin/sh” argument on the stack (only works on 32 bits). A more interesting subclass of code-reuse attack is named return oriented programing (ROP). It uses the fact that the control flow can jump in the middle of any kind of existing function to only execute what is needed. For instance, calling system(”/bin/sh”) on 64 bits means first loading the ”/bin/sh” address in the right register and then jump to system(). Loading a register like rdi needs only a `pop rdi` instruction followed by a return since we assume the attacker can take control of the content of the stack. Therefore, an attacker needs only to jump to a pop instruction into rdi followed by a return address. By writing the address of system() on the stack, the ”gadget” (pop instruction and return) will simply load whatever the attacker wrote at the top of the stack into rdi before returning at the system() function. For our paper, we execute a slightly more complex call than the system call (execve) which leverages the same principals as those we just described. This paper explain how we used this type of attack to get privilege escalation on code with a simple vulnerability, the format string vulnerability. This vulnerability is not used to overwrite a return address. As a result, the canaries added at compiler time do not prevent our attack in any way. Nevertheless, we had to disble the ASLR protection to keep this exploit easily understandable.

III. Design

A. Format string vulnerability

The core of our vulnerability exists in that in the fuction Peanut we pass an arbitrary buffer of user input (gathered using fgets) to the function printf (line 9 of our c code: `printf(buffer)`). This introduces a format string vulnerability that will allow us to ”Write What Where” using the special format string %n. In our exploit, we will use this vulnerability to overwrite the Global Offset Table (GOT).

B. Exit and the Global Offset Table

Our vulnerable function, Peanut contains a call to Exit before the end (line 11 of our c program: `exit(1)`) . This means that we cannot overwrite the return pointer since the return pointer will never be called (we will exit before then). However, exit will be called from the Global Offset Table. This means that if we can overwrite the address to exit in the GOT, we will be able to redirect the programs control flow

C. execve

In order give a big ”Look over here” hint as well as make our exploit easier, we have embedded the arguments to call execve onto the stack in our main function (lines 22-24). This will make it easier to call bin shh from execve. We believe that it could also be called by embedding those arguments in the buffer.

Note that we did explore calling mprotect to make the stack executible to call our shell code instead of calling execve directly using gadgets. While we got this working in GDB, we could not do so outside of the debug environment as mprotect needed to know the exact start of the stack page and it's exact size in order to work correctly, and we were unable to find this information with our avaliable program. This is something we would like to attempt again in future work.

IV. Implementation

The following steps must be followed to get the privilege escalation:

1) Start by compiling the code with the following command: `gcc -o peanut peanut.c`
2) Disable ASLR with the command `echo 0 | tee /proc/sys/kernel/randomize_va_space`
3) The exploit is printed in the `exploit` file. Just run `(cat exploit; cat) | ./peanupower` to run `bin/shh`

V. Exploitation

The buffer we used to execute `execve("/bin/shh")` has the following contruction:

```
+---------+  ^
| %x & %n |  |  stack
| three   |  |  grows
| times   |  |  this way
+---------+  +
|   ROP   |
|  chain  |
+---------+
| padding |
| optional|
+---------+
|  exit   |
|addresses|
+---------+
```

We explain step by step how to construct the exploit in this section.

A. *Format String Exploit*

The core of our exploit involves the fact that printf prints a buffer of user input. This is a vulnerability which allows us to use special string formatting characters as input in order to create malicious affects on the system. In particular, we can use %x and %n.

The special format specifier %n allows us to print the number of characters printed into a particular location. This will allow us to have a write what where vulnerability (woo hoo!). In our case, we will use this to overwrite the value of the global offset table so that when exit is called, it will eventually call our overwritten address instead. In order to do that we first need to find out what address we need to overwrite. We do that by debugging through our code in order to find the location of the global offset table. We see that the function called is `<exit@plt>`, which jumps directly to another address, which is the place in the GOT which store the real address of exit(). In our case, this address is `0x601030`. This is what we need to overwrite with our first gadget. We can put this value into our buffer and then reference it as our write location.

We can then try overwriting that writing location by printing (using printf) a number of blank spaces equal to the address, and then writing that as a hexidecimal representation (%x) to our specified address using %n. But wait, addresses in libc are really big, which is a lot of printing of extra characters. When we tried this as a single call it did not work. As a result, we decided to break up our write into 3 segments where we write 4 hexidecimal values each. Note that we do not have to write 4 of the values since the topmost values are 0s both for the original address, and for our gadgets in libc. In order for this to work, however, we had to print the values in order from least to greatest, so that we were printing progressively more spaces and the moving that value of already printed spaces into the correct places.

When executing the %n specifiers which point to the exit addresses, the printf() function will overwrite the entry in the GOT table. We needed to write the address `0x7ffff7b49c0e` at the address `0x601030` (see subsection on exit address). We write this adress in three times, in the following order:

1) `0x7fff` at `0x601034`
2) `0x9c0e` at `0x601030`
3) `0xf7b4` at `0x601032`

Since %n write the number of printed characters since the beginning of the execution of printf(), we need to write the address in the increasing order of their values. Because the position of the addresses of exit on the stack where from the 67th to the 69th and we only want to write 2 bytes, then the exploit must start with:

```
exploit = ""
exploit += " %{}x ".format(0x7fff)
exploit += " %67$hn "
exploit += " %{}x ".format(0x9c0e - 0x7fff)
exploit += " %68$hn "
exploit += " %{}x ".format(0xf7b4 - 0x9c0e)
exploit += " %69$hn "
```

A few change in the numerical values must be performed to take in account that we print spaces with the specifiers.

B. *Poppin'*

But what do we overwrite it with? At first, we tried to do a simple pop to any register and then a return (as the exit command itself with push 0x1 onto the stack). We were then put our shell code before our printf writing vulnerability in our code, so that when our gadget returned, it would immediately start executing the rest of our shell code gadgets. However, when we did that, we immediately went from correctly overwriting the address to nothing happening. Upon further reflection this was because our gadget addresses had null bytes and printf stops printing when it encounters a null terminator. What a pain!

So instead of doing just a simple pop, we instead put the shellcode AFTER our printf exploit (although before the reference address for our overwriting since we did not need that inside the exploit). We were then able to find a gadget which performed seven consequtive pops and then returned. This allowed us to pop off our printf exploit call and the 0x1 pushed onto the stack from the exit code so that we would then return and grab the next gadget address as our return pointer.

C. Execve call

Once we successfully grabbed control of the program, we then had to use gadgets to call execve since we had a non-executable stack. Since we already had the parameters we needed in main, it was easy to grab the addresses for the parameters that execve needed. We found gadgets to pop them into the correct registers and placed them onto the stack such that they would be correctly popped off. We then called execve on bin shh in order to launch a shell.

By writing the `execve("/bin/shh")` code in C and looking at the assembly, we list the content of the registers we need to set up.

1) rax: execve code, 0x3b.
2) rdi: address of "/bin/shh" string, needs to be followed by a null byte.
3) rsi and rdx: 0x0.

The implementation of the ROP chain in the exploit buffer is as followed:

```
exploit += <pop rax address>
exploit += 0x000000000000003b
exploit += <pop rdi address>
exploit += <address of "/bin/shh">
exploit += <pop rsi address>
exploit += 0x0000000000000000
exploit += <pop rdx address>
exploit += 0x0000000000000000
```

We used the struct package and its pack() function in our exploit to implement this ROP chain.

D. Padding

Padding to cover the whole buffer is optional but useful when creating the exploit. If we need to add something before the exit addresses, their position will change and we will need to start again, determining their position to point at it again. With a padding, the position of the exit addresses will stay the same whatever we append to the exploit buffer before padding. We implemented the padding as follow:

```
def pad(s):
return s+X*(buffer_size - len(s)
             - exit_addresses_size)
exploit = pad(exploit)
```

E. Mprotect

Initially we had not planned to launch execve via gadgets, but instead had planned to call mprotect to change the permissions on the stack to allow execution. We managed to place the gadgets on the stack in a similar way to our execve call, and then had a jump to the middle of our NOP slide for shellcode execution. Unfortunately, while this worked perfectly within GDB, outside of GDB we were not able to get this working. This is because GDB allows you to see the exact location of the stack and it's size which we needed for the mprotect call. Despite turning off ASLR, the location seemed to be different outside of the stack, and while we tried various ways in order leak the information through our printf statement, we were not able to get an accurate enough location for the stack in order to get mprotect to work. As a result, this was abandoned in favor of simply using gadgets to call execve. We hope to explore this more in the future.

VI. Conclusion

We presented in this work how to use a format string vulnerability to overwrite the global offset table to get control of the flow of the program. We also demonstrate how to use this flow to chain ROP gadgets which can be used to execute arbitrary code like the execve("/bin/shh") we needed here. This paper also introduce how to use the mprotect() function to make the stack executable again, in order to be able to run a hand-made shellcode, like we did in the previous project.