PeanutPower: CS557 Project 2

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

Brittany Lewis
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. 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. 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. 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. 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.

II. Exploitation

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. Since printf executes a call to the PLT to jump to the GOT, we can debug through that call where we find an address at 0x60130. 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.

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.

D. 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.