Project2 Report

Jay Challangi

April 2023

1. Simple Command Line Buffer Overflow

Vulnerability

In this exploit I specifically exploited line 21 system(p) where it does the system call on string p, which is a concatenated string of the ls command with user input.

Explanation

My attack strategy was that the string passed can call as many commands as passed including the first ls command. All we needed to do to get the system to do multiple commands was separate our commands in the input with a semicolon. By putting random text followed by a semicolon followed by the command we want to run, in this case /bash/sh, we are able to get the program to run both ls and our command when the system command in target1.c is called on p.

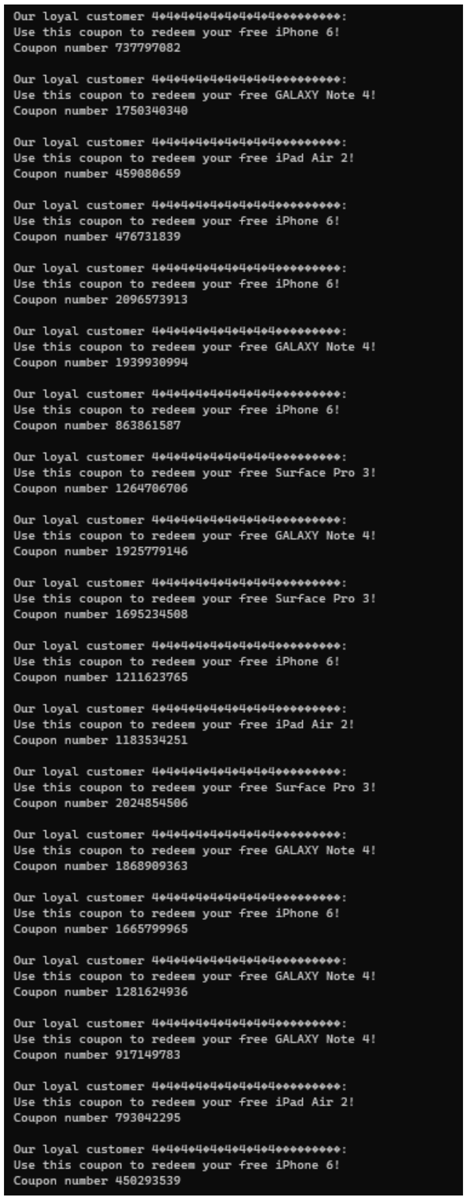
2. Buffer Overflow To Rewrite a Return

Vulnerability

In this exploit I specifically exploited line 12 where strcpy(name,arg) copies the user input in arg (which was argv[1] in main) to the name array. This would be a buffer overflow attack.

Explanation

In this exploit I overflowed the container for name past its intended size all the way to the return address containing the instruction pointer pointing to main (when coupon is called its instruction return address is the line in main where coupon was called). I overwrote the return pointer to point to the address of the coupon function. In addition to this, to make the coupon function be called an unlimited number of times, after the initial address of coupon in the buffer that is used to overwrite the return pointer which was to main, I include the address of coupon again(the function for the second coupon call to go to after its done, the first coupon is called in main), followed by the address of argv[1] twice which ensured that the next function uses argv[1] as the argument for the coupon fucntion. Since coupon is always called by argv[1] it will be called an unlimited number of times as the name array is overflowed each time by the same input that caused the initial repeated call of coupon.



```
Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPhone 6!
Coupon number 737797082

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free GALAXY Note 4!
Coupon number 1750340340

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPad Air 2!
Coupon number 459080659

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPhone 6!
Coupon number 476731839

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPhone 6!
Coupon number 2096573913

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free GALAXY Note 4!
Coupon number 1939930994

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPhone 6!
Coupon number 863861587

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free Surface Pro 3!
Coupon number 1264706706

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free GALAXY Note 4!
Coupon number 1925779146

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free Surface Pro 3!
Coupon number 1695234508

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPhone 6!
Coupon number 1211623765

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPad Air 2!
Coupon number 1183534251

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free Surface Pro 3!
Coupon number 2024854506

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free GALAXY Note 4!
Coupon number 1868909363

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPhone 6!
Coupon number 1665799965

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free GALAXY Note 4!
Coupon number 1281624936

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free GALAXY Note 4!
Coupon number 917149783

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPad Air 2!
Coupon number 793042295

Our loyal customer 4◆4◆4◆4◆4◆4◆4◆4◆4◆4◆◆◆◆◆◆◆◆◆◆:
Use this coupon to redeem your free iPhone 6!
Coupon number 450293539
```

Figure 1: The Output of the Exploit

Defense

The attack did not work against the edgar machine. The edgar machine detected that we were stack smashing (rewriting the return address) and aborted the program after the initial call of coupon.

One defense that would help prevent this exploit would be the use of compiler-specific protections. GCC specifically can help prevent stack smashing by compiling the c code with the "-fstack-protector" option, which inserts canaries into the program at the compile-time. An advantage of compiler-specific protections do not require any modifications to the program's source code or the operating system, thus making them easier to implement. A disadvantage of compiler-specific protections is that they may introduce some performance overhead, as additional code needs to be inserted to implement the protections. Additionally, this defense is not foolproof to attacks and should be used with other defenses.

Another defense that would help prevent this exploit would be safe programming practices. This would mean writing the code such that it checks for buffer overflow vulnerabilities and uses the safe versions of functions such as "strcpy_s" instead of "strcpy" for strings. An advantage is that safe programming practices can help prevent other types of security vulnerabilities, such as integer overflows, format string vulnerabilities, and race conditions. A disadvantage of safe programming practices is that implementing safe programming practices can require additional time and effort, especially for developers who are not familiar with the techniques. This can result in longer development cycles and increased costs.

3. Return to libc

Layout

StackTop
Arg2 char ** argv (4bytes)
Arg1 int argc (4bytes)
Return IP (4bytes)
Saved EBP (4bytes)
char traffic [64] (4bytes)
int i (4bytes)
int j (4bytes)
int len (4bytes)

Vulnerability

In this exploit I specifically exploited line 17 where strcat(traffic, argv[i]) will concatenate the user input in argv[1] to the char array traffic. This overflows the container of traffic as traffic is only of size 64 bytes (buffer overflow in the strcat function itself). This a return to libc attack as after overflowing the buffer the strcat function will be caused to return to system (a libc function) instead of its proper return location (in is_virus)

Explanation

In this exploit I overflowed the container for traffic past its intended size in the strcat function all the way to the return address containing the instruction pointer pointing to is_virus (when strcat was called its instruction return address is the line in is_virus where it was called). I overwrote the return pointer to point to the address of the libc function system. The libc function system needs a return address (in my case I did a NOPE slide for the address) as well as an argument. The argument needs to be the pointer to the string "/bin/sh" to get the system to call the shell. The way I stored that string is by saving it to the environment and in the payload where the argument needs to be is the address of the string (found through looking at the environment variables through core dumps).

Defense

The attack did not work against the edgar machine. The edgar machine detected that we were stack smashing (rewriting the return address) and aborted the program after the initial call of coupon.

One mechanism that is stopping this is stack protection systems implemented by the operating system (in this case) that is checking for stack smashing thought the use of canaries.

4. Format String Attacks

Layout

StackTop
Arg2 char ** argv (4bytes)
Arg1 int argc (4bytes)
Return IP (4bytes)
Saved EBP (4bytes)
char user_input[100] (4bytes)
int *secret (4bytes)
int int_input (4bytes)
int a (4bytes)
int b (4bytes)
int c (4bytes)
int d (4bytes)

Inputs

To crash: 1, ”A” *124
Any string longer than 123 characters. It crashes because the return pointer is overwritten

Address of the secret[0]: 1, %11$p
It is the same address for secret[0] as in gdb

Value of the secret[0]: 1, %11$s
It outputs D which is the hex equivalent of 0x44

Address of the secret[1]: 134520844, %10$p
This is the integer value of the address of secret[0]+4 as the next element of the secret array is address of secret[0]+4. That is how arrays work in the heap

Value of the secret[1]: 134520844, %10$s
I found the value of secret[1] the same way i found the value of secret[0]

Modify both: 134520840,11111%10$lln
The n option will rewrite the 4byte value. If it is just n it will only modify secret[0]. Because secret[0] is next to secret[1] with lln we modify 8bytes with the value.

Question Answers

ASLR would make this format string attack harder as we could not be able to use gdb to confirm the address of secret and would have to guess and check every value for it. It generally does make it harder for format string attacks to work as an integral part of some of those attacks are finding the addresses of variables and so on.

One operating defense against this attack could be FormatGuard. It adds runtime checks to the format string functions to ensure that the format string and arguments match. If the format string does not match the arguments, or if the arguments are of the wrong type, FormatGuard will terminate the program and generate an error message.