Casper CTF: Solutions

Sanchari Bandyopadhyay r0919751

February 16, 2023

1 Overview

Level	Password	Time spend
3	6chZrfRWNfBRCP4Sya91vieNWVXrRoRv	0
4	HikOeJRoG2kYCI1QrHBf0bouflFWStPy	4
5	XRTOmu0ToSDrWsPMv7UFPqRcDJFUVf6E	7
6	vb56zXrIsjRPrpM81f5EsqnAjBKYnft8	3
61	8NuwK9iWoCpNlmLNVKmagieWe0j0StHl	4
62	GG2RieHeoogpMJF1qFyQhfb8Vm7SUh6P	0
63	oADxWU6GKYmesTfKRiihFr3wK8bMnFxM	4
7	UlzsukgATPSjI4eiMU8JkmAi3LKBvLMP	6
8	42PX6QWbUgN3EeEWrBkI1KJZSip6yCQV	2

2 Casper4 solution

2.1 Description

Casper4 declares a struct containing a buffer and a function pointer. It takes in the name of the user as a command line argument and uses the vulnerable function strcpy to copy the input into a buffer stored in a struct. Then it calls the function located at the address of the function pointer.

The important parts in this program, all allocated on the stack, are:

- The buffer of 775 characters,
- The function pointer pointing to greetUser(),

2.2 Vulnerability

This program contains a vulnerability because the strcpy call does not do a bounds check for the length of the string copied into the buffer. Hence, the adversary may provide a string longer than the bounds of the buffer to corrupt the contents of memory beyond the buffer address.

2.3 Exploit description

As the stack is executable, we will insert the shellcode at the start of the buffer and fill in the rest of the buffer with padding. As the function pointed to by the *fp is called next, we would want to find the address of the *fp and overwrite its contents with the address of the buffer so that the shellcode is executed. As both the buffer and *fp are declared in a struct, they should have adjacent memory addresses making it easier to overwrite the *fp content.

The first part of the input is the shellcode and it will be placed at the start of the buffer. The next part of the input is padding to fill in the memory until the function pointer address. The last part is the address of the buffer - 0x08049840 which overwrites the original value at the function pointer , &greetUser.

2.4 Mitigation

In the source code, we should use a safe function such as 'strcpy_s' which does a bounds check for the string being copied and the buffer size. This ensures that there is no buffer overflow. Finally, the OS/compiler, we should disable executable stack. This ensures that there is no shellcode injected into the buffer, and no memory beyond the buffer is corrupted.

3 Casper5 solution

3.1 Description

Casper5 declares 2 structs, user_t and role_t. user_t contains a buffer of size 775 and a pointer that points a role_t. It asks for an input of the user and uses the vulnerable function strcpy to place the user input into the buffer inside user_t. The default role assigned to any user has an authority value of 0, but if the authority value is set to 1, the user will be an admin and the flag will be printed.

The important parts in this program, all allocated on the stack, are:

- The buffer of 775 characters,
- The *role pointer pointing to role_t,

3.2 Vulnerability

This program contains a vulnerability because the strcpy call on does not do a bounds check for the length of the string copied into the buffer. Hence, the adversary may provide a string longer than the bounds of the buffer to corrupt the contents of memory beyond the buffer address.

3.3 Exploit description

The stack is non executable and stack canaries are enabled, hence we cannot use shellcode, or overwrite the return address. As this is a data-only attack, we will have to modify the value that thisUser.role.authority points to. The initial approach I took was to try to fill in the buffer with '1' and overwrite the *role pointer to an address in the buffer. However, it is a char buffer and thisUser.role.authority is an integer. Hence, an input of 0x00000001 would not be stored in the buffer as null characters 0x00 will be skipped.

In the next approach, using gdb find, I tried to find `0x00000001` already present in memory and overwrite the value at thisUser.role.authority with the address of the `0x00000001`. The address of `0x00000001` seemed to change if the exploit was ran from different directories or commands, hence with some brute force, when run from the verify_tarball command in an empty /tmp/folder directory, the address of `0x01` was (0xbfffef68)

The address of thisUser.role appears after 776 bytes from the start of the buffer and it is overwritten with the address `0xbfffef48` which is 32 bytes less than the address of `0x01`, `(0xbfffef68)`. As thisUser.role.authority's address is 32 bytes after thisUser.role, thisUser.role.authority will point to `0xbfffef68`.

3.4 Mitigation

In the source code, we should use a safe function such as strcpy_s which does a bounds check for the string being copied and the buffer size. This will prevent a data-only attack as no memory beyond the contents of the buffer can be corrupted. OS level mitigation strategies can be bypassed as no shellcode is injected, and the return address is not modified.

4 Casper6 solution

4.1 Description

Casper6 is a simple program which uses the vulnerable function gets() to copy the user input into the buffer.

The important parts in this program, all allocated on the stack, are:

- The buffer of 775 characters,
- The return address,

4.2 Vulnerability

This program contains a vulnerability because the gets() call does not do a bounds check for the length of the string copied into the buffer. Hence, the adversary may provide a string longer than the bounds of the buffer to corrupt the contents of memory beyond the buffer address.

4.3 Exploit description

Stack canaries and non-executable stack are disabled, hence we may use shellcode and do a simple buffer overflow attack to get the flag. Similar to Casper4, we will need to store the shellcode as input in the buffer, fill the rest of the buffer with padding, and then overwrite the return address to the address of the buffer.

Using gdb, I created a breakpoint in the function greetUser(), so that I could use info frame, and find the address where the return address was store (eip). The return address is 787 bytes after the start of the buffer. Since the address of the buffer might be dynamic and differ by about 10 bytes, I placed NOPs at the start of the buffer and overwrote the return address to be deep inside the buffer. This ensures the shellcode will be executed.

4.4 Mitigation

In the source code, use a safe function such as fgets() which does a bounds check for the string being copied and the buffer size. In the OS level, we should disable executable stack and enable stack canaries. Together, these strategies will prevent modification of memory beyond the buffer, prevent execution of any shellcode, and detect if the return address has been modified.

4.5 Advanced levels

4.5.1 Casper61 Solution

4.5.2 Description

Casper61 differs from Casper6 as it checks for NOP bytes in the buffer. However, in the loop, it stops the checking as soon as it encounters a null byte.

4.5.3 Exploit description

Hence, we can prevent the check from taking place by putting a null byte at the start of the buffer. Then we can proceed to do the attack as per Casper6, by placing a NOP sled and the shellcode in the buffer, and overwriting the return address to point to somewhere deep in the buffer.

4.5.4 Casper62 Solution

4.5.5 Description

Casper62 differs from Casper6 as it resets the environment before running the program.

4.5.6 Exploit description

As I had not set any environment variables in my exploit for Casper6, I was able to perform the attack on Casper2 in the same way I did for Casper6.

4.5.7 Casper63 Solution

4.5.8 Description

Casper63 differs from Casper6 as it checks that all the bytes in the buffer are ascii characters. Once it detects a non-ascii character, the program exits.

4.5.9 Exploit description

For this exploit, we cannot place any shellcode inside the buffer, as the program will exit. However, we can place the shellcode after the buffer as there are no stack canaries. Hence, I filled the buffer with ascii characters until the address where the return pointer is stored. I then overwrite the return pointer with an address deep after the end of buffer. Then I insert a NOP sled, and the shellcode. Once the program returns to the address we have inserted, it will run the NOPs and the shellcode, giving us the flag.

5 Casper7 solution

5.1 Description

Casper7, similar to Casper6, has a buffer of size 775 and uses the vulnerable gets() function to copy the user input into the buffer. However, the stack is not executable.

The important parts in this program, all allocated on the stack, are:

- The buffer of 775 characters,

5.2 Vulnerability

This program contains a vulnerability because the gets call does not do a bounds check for the length of the string copied into the buffer. Hence, the adversary may provide a string longer than the bounds of the buffer to corrupt the contents of memory beyond the buffer address.

5.3 Exploit description

The stack is non executable, hence we cannot use shellcode. However we can overwrite the return address and use predefined C library functions to execute a shell. We can execute the return-to-libc attack.

First I exported and environment variable, MYSHELL = '/bin/xh'

Similar to previous exploits, we have to find the offset of the return address from the start of the buffer. Once we find that, we can overwrite the return pointer to the address of the system() call. This will execute the system call. The next address should be the return address of the system call, and it is to be overwritten with the address of the exit() call. The next address value will be overwritten with the address of MYSHELL.

Using the debugger, we are able to find that the address of MYSHELL often changes but stays within the range of `0xbffffd0A` to `0xbffffdFF`, and hence in our bash script, I wrote a loop to brute force and try a range of addresses for MYSHELL to be executed. Using gdb, we can also find the address of the system() and exit() call.

5.4 Mitigation

In the source code, we should use a safe function such as fgets() which does a bounds check for the string being copied and the buffer size. In the OS level, stack canaries should be used to detect return address modification. This ensures that a return-to-libc attack does not work, as no memory beyond the buffer can be corrupted and the return address modification will be detected.

6 Casper8 solution

Casper8 is a program that takes in the user input string as a command line argument. It then calls the vulnerable function sprintf() with the user input string, to print a greeting.

The important parts in this program, all allocated on the stack, are:

- The buffer of 775 characters,

6.1 Vulnerability

This program contains a vulnerability because the sprintf() call does not do a bounds check for the length of the string copied into the buffer. Hence, the adversary may provide a string longer than the bounds of the buffer to display or corrupt the contents of memory beyond the buffer address.

6.2 Exploit description

We may use the format string vulnerability to view and modify contents at different memory addresses.

First I inspected the binary to find the address of the variable isAdmin. It was `0x0804999c`. Then I found out at what offset from the buffer the user input argument is stored using a long string of '%x'. Using the format string %¡num¿$n to write to the isAdmin address. We can write any value other than 0 and it will return true.

6.3 Mitigation

We may do a manual check that the user input is only ASCII characters so that format strings may not be entered as input. We should use safer functions such as snprintf() or asprintf() so that adversaries are not able to overflow the buffer and view memory contents.