Casper wargame Solutions

Francesca Del Nin
student number: 0734630

December 2018

1 Overview

Level	Password	Time spent
casper4	zssEylQhyfOdX0H7OOKFxEXG0iY9Y7PL	
casper41	qQftts3uK2DjrNxfEPzXTyOpB20vMakY	
casper6	l1BcWzPZaWMGOM0kaWimJmmT40KmDE9o	
casper60	EoWZv5u7pnBkchLK30vXhiyH1sDexpCB	
casper8	AAqtUL09LWefJlJFvTg0SpVg0j89DGe2	
casper80	5CS80xbUrxgjbrU3BBS5YslY00qoUd5J	
casper81	yMxdglE84E0NpXIXXRcukr8uEWkIHE6w	
casper82	MABIQpR4omrMCmdc7PwxbRNm5EPHzYX5	
casper10	BpI02agB8PmOKDI1LRvMeQJWi2GrzgWA	

2 Casper4 solution

2.1 Description

Program casper4 takes an input from the user and prints a greeting and the input from the user. If no input is provided it prints an explanation on how to use the program and exit.

2.2 Vulnerability

This program is vulnerable because it calls `strcpy` function passing the (char) pointer `s` without doing a bound check on the length of the variable pointed, so it can be longer than 666 byte (or the 674 allocated), which is the length of the `buf` array in which the content will be copied.

2.3 Exploit description

To exploit this level I first needed to know how many characters fill the buffer and overwrite the return address of `greetUser`. To find this number I used gdb, put a breakpoint inside the `greetUser` function and runned the program, the address of the buffer is:. Then, by using `info frame` command I'm able to see at which address is stored the eip register where the return address is stored. The distance between the two is 678 byte, so by adding another 4 byte I'm able to make a buffer overflow attack.

I provide as input a string of 682 byte in total composed as following:

1. 100 NOP,
2. the shellcode (21 byte),
3. some padding, in this case "A"'s but can also be NOP (557 byte),
4. the address of the buffer, retrieved through gdb by placing a breakpoint inside the `greetUser` function and using the command `p &buf` on the server.

In this way I fill the whole buffer and overwrite the return address (and the frame pointer too) of the function, so when the execution of `greetUser` is done the instruction inside the buffer (so the shellcode) will be executed instead of returning to main.

The NOP instructions at the beginning of the buffer are to make sure that even if the address of the buffer is not precise (due to the program being run inside gdb), the shellcode will be executed because is preceded just by NOP instructions.

2.4 Mitigation

2.5 Advanced level casper41

This exploit works also with casper41 because it only checks for environment variables which are not used for the exploit. I just need to update the address of the buffer, using gdb on the binary file of casper41.

3 casper6

3.1 Description

This program prints a greeting to the user when provided an input, it uses a struct composed by a buffer array and a function pointer.

3.2 Vulnerability

This program is vulnerable because it is possible to overwrite the function pointer, which is store just above the buffer. The struct is not stored in the stack because it's a global variable and it's stored in the data segment of the memory, so even if the stack is non-executable it's still possible to run an overflow attack.

3.3 Exploit description

3.4 Mitigation

4 casper8

4.1 Description

This level prints a greeting to the user as the other levels, but in this case the stack is non-executable so it's not possible to do a stack based buffer overflow.

4.2 Vulnerability

In this case the vulnerability is given by the use of the library

4.3 Exploit description

To make this exploit work I first looked for how many input characters make the program go in segmentation fault, then I verified that I was overwriting also EIP register using the "info frame" command after the execution of strcpy inside greetUser and found out that I need a total of 682 byte to complete overwrite the return address of greetUser.

Then I searched for the address of the `system` function (which execute a shell command) inside libc, which is loaded because it's included in the program. I used gdb command `p &system` while running the program and I found that the address is 0xb7e62310. Since I want to pass as argument to this function the string "/bin/xh", which is not present inside the loaded library, I created a new environment variable (through export MYSHELL=/bin/xh). To find out the address in which is stored I used the command `x/s *((char **)environ)` and went through the variables until I found MYSHELL at *((char **)environ+19) at the address 0xbfffff3f, so to make it point just to "/bin/xh" (without "MYSHELL=") the address is 0xbfffff47.

To make it a cleaner attack I also searched for the address of the `exit` function inside the library to pass it as return address of the system function. I used the same command as above and found it at 0xb7e55260.

To make the exploit work I passed a string composed as following:

- 682-4=678 A's;
- `system` function address (endian encoded);
- `exit` function address (endian encoded);
- the address of the string "/bin/xh" (endian encoded).

This attack worked inside gdb but not outside because the address of the environmental variables can change a bit, the error displayed was telling "bin/xh: not found" so the string passed was missing a "/". Changing the address of the string to 0xbfffff46 did the trick.

4.4 Mitigation

This attack was possible even if the stack is non-executable, so it requires additional measures.

Adding canaries can help detecting if a return address of a function id being overwritten but if the attacker is able to read this value it can overwrite it with the same value.

Enabling ASLR (Address Space Layout Randomization) can make it more difficult because the location of the stack, heap and library can vary each time the program is launched, so it became difficult to retrieve the address of the functions to which an attacker wants to jump.

4.5 Advanced level casper80

The same approach work also for this advanced level because it only checks for NOP in the input. To make it work I checked again the distance from the buffer to the return address of `greetUser` function, this time it has changed to 682 byte (so 686 to overwrite it). I also changed the address of the string because the address of the environmental variable MYSHELL changes a bit, to make it work I used 0xbfffff44. To get this value I read the error message displayed with the address used in the other exploit and there where two characters missing.

4.6 Advanced level casper82

The same approach works also for casper82, because it checks for ASCII characters just for the length of the buffer, so if I fill the whole buffer with A's and only after use the ascii characters to overwrite the return address the attack will still work. The exploit in this case is exactly the same as casper80.

5 casper10

5.1 Description

This program prints a greeting to the user if it's provided an input string, then checks for the admin flag and if it's true it launches a shell.

5.2 Vulnerability

This programs contains a format-string vulnerability, this makes it possible to change the value of the flag even if canaries and non-executable stack are enabled.

A format string is an ASCIIZ string that contains text and format parameters, if the attacker provides in input format parameters (like %s) instead of a string it can read and write in memory, for example the parameter %x read data from the stack and prints it

5.3 Exploit description

To make this attack work I have to read from the memory the location of the `isAdmin` variable and use the string vulnerability to change it (any value beside 0 will work since it's a binary variable).

To do so I will use the %n parameter which writes the number of bytes already printed, into a variable we can choose, in this case `isAdmin`. Before being able to do so I need to move the stack pointer to a location where there is the address of `isAdmin`, for example the beginning of the buffer, of which I can provide the content.

To move at the beginning of the buffer I use %x, which reads from the stack and goes towards higher addresses (so towards the top of the stack), the buffer is stored in higher address because is pushed into the stack before the call to the function printf.

Filling the buffer with some 'A"s characters and some %08x parameters (08 makes it print in exadecimal) will print back the A's (x41), at this point I know the number of %x parameter I need to reach the beginning of the buffer.

At this point if I write the address of the variable instead of the first four 'A's and add a %n parameter I'm able to write some value in that address, or in other words change the value of the flag.

The string provided as input is:

- the address of the `isAdmin` variable (retrieved through gdb, little-endian encoded),
- nine %80x parameter to reach the beginning of the buffer,
- %n to write in the address.

5.4 Mitigation