Casper wargame Solutions

Francesca Del Nin
student number: 0734630

December 2018

1 Overview

Level	Password	Time spent
casper4	zssEylQhyfOdX0H7OOKFxEXG0iY9Y7PL	
casper40	hcRzYRX24hMUQfGN4comHv8pXgl7rgiK	
casper41	qQftts3uK2DjrNxfEPzXTyOpB20vMakY	
casper5	LzroSg7w54LlHxKZbHVEGYj2UPs1WEUG	
casper50	SuzPtkbto6T8w0H43MQimqOoH7ntIoVh	
casper51	TOv1elxDKtaFgGfWTyyLQ2uzivxI36X4	
casper52	rWhAgxz2U89eRx57LyCmEWunbCW39fAO	
casper6	l1BcWzPZaWMGOM0kaWimJmmT40KmDE9o	
casper60	EoWZv5u7pnBkchLK30vXhiyH1sDexpCB	
casper61	K14irUGErx7IC3Gvh5ZdO5sIKt5eQCde	
casper62	qe3YqqaQSkvJhTEMxt68uUwemCc9d1WI	
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 stored 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

To exploit this level I get the address of somedata.buffer and somedata.fp through gdb, in this way I know how many characters I need to reach the function pointer and overwrite its content (668 bytes).

The string I pass as input is composed as follows:

1. some NOP instruction
2. shellcode (21 bytes)
3. some more NOP to fill the buffer
4. the address of the buffer

Since the address of the buffer running the program inside gdb and without it can vary a bit, the NOP instructions at the beggining are necessary to make sure that the exploit works even if the address of the buffer varies a bit.

3.4 Mitigation

One way that can mitigate this attack is using ASLR(Address) which makes more difficult retrieving the address of the buffer since it will vary at every execution of the program.

3.5 Advanced level casper61

This program adds an additional check at the environmental variables (that can be used to store the shellcode), the same approach of casper6 can be used also here, I just changed the buffer address.

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, but canaries are not enabled.

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 the whole EIP register (where the return address is stored) using the "info frame" command after the execution of strcpy inside `greetUser` and found out that I need a total of 682 byte to completely 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 using make. 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, since the exact address can vary a bit the variable contains some spaces and only then "/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).

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.

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 environmental variable (endian encoded).

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 has been 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 becames 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).

4.6 Advanced level casper81

This level checks for the environmental variables so to exploit it I changed the approach. The stack is non-executable but it's still possible to use the stack to store the string we want to pass as argument to the `system` function.

To make it work I filled the buffer with some spaces, the string, some other spaces and then the address of the system function, followed by the exit function and then the address of the buffer.

Before using the spaces I filled the buffer with some A's before /bin/xh and some B's after, in this way I was able to understand (by reading the error) if I was pointing somewhere after the /bin/xh string.

To make it possible to store spaces inside the buffer I used the bash script.

4.7 Advanced level casper82

The same approach of casper8 works also for casper82, because it checks for non-ASCII characters just for the length of the buffer, so if I fill the whole buffer with A's and only after use the non ascii characters to overwrite the return address the attack will still work. The exploit in this case is exactly the same as casper80 with environmental variables.

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 (string is terminated by the 0 character) 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 I can store the address of `isAdmin`, for example the beginning of the buffer.

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

To make this program more secure it's better to use a different function instead of `printf`, also using ASLR will make more difficult to retrieve the address of the variable we want to write to.