# 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 exits.

## 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 found is 0xbffff146. 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 address is 0xbffff6bc. 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 to fill the buffer and reach the return address (557 byte),
4. the address of the buffer.

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 it is preceded just by NOP instructions. To make this attack work I had to change a bit the address of the buffer, I put a slightly higher number to make sure it's pointing inside the buffer even if the buffer moves a bit(final address: 0xbffff186).

## 2.4 Mitigation

Different things can be done to mitigate this attack: checking the length of the string copied to the buffer is a first thing that can void this attack. Another thing that can be done is making the stack non-executable, so even in case of a buffer overflow no code inside the stack will be executed, but the overwrite of the return address is still possible. To make it more hard it's possible to add canaries that make easier to detect this kind of attack. Another measure that can help is enabling the ASLR (Address Space Layout Randomization) that makes it harder to get the correct address of the buffer since it changes at every execution of the program.

## 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.

## 2.6 Advanced level casper40

This level has an additional check compared to level4: it looks for NOP instructions inside the string provided in input. To exploit this level I used a different approach: instead of storing the shellcode inside the buffer I stored it in a new environmental variable (together with some NOP instruction).

In this way I am able to fill the buffer with A's and the address of the variable (which does not contain
x90)

Also the distance between the buffer and the return address is changed, now I need 682 byte to reach it.

So the string provided in input is:

1. 682 A's
2. the address of the environmental variable

To get the correct address of the environmental variable I used gdb (`x/s *((char **)environ))` which shows the environmental vars and their address. I found out that the address is 0xbffff8b4. To make this exploit work I changed a bit the address to be sure that it points between the NOP's.

As last thing I remove the environmental variable.

## 2.7 Advanced level casper42

This level checks for non-ascii characters inside the buffer but it does not check for the entire length of the string provided. To exploit this level I used the same approach of level 40 since it fills the buffer with A's characters and only after (more than) 666 byte there is the first non ascii character(the address of the environmental variable).

# 3 casper5

## 3.1 Description

## 3.2 Vulnerability

## 3.3 Exploit description

## 3.4 Mitigation

# 4 casper6

## 4.1 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.

## 4.2 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.

## 4.3 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 an reach the function pointer
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 beginning are necessary to make sure that the exploit still works.

The address of the buffer appears to contain the
x00 value, so I changed the last byte to 01 otherwise the input

## 4.4 Mitigation

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

Also checking the length of the string before copying can be countermeasure to this attack.

## 4.5 Advanced level casper61

This program adds an additional check at the environmental variables (that can be used to store the shellcode), which I did not use in exploiting level6, so the same approach can be used. In this case I changed the address of the buffer since it has changed to 0x8049860.

## 4.6 Advanced level casper60

This levels checks for NOP instruction inside the string provided as input. To exploit this level I stored the shellcode inside an environmental variable and filled the buffer with A's and the address of the variable.

Together with the shellcode I stored also some NOP instruction to make it easier to get an address that makes the attack work.

To get the address I exported the variable and used gdb to get the address, then I changed the address to a slightly higher one to make sure it will point inside the NOP instructions.

The string provided as input is:

1. 668 A's
2. the address of the environmental variable

## 4.7 Advanced level casper62

The same exploit of casper60 works for casper62 because this level checks for non ascii character but only for the expected length of the buffer (666 bytes) and not for the entire string provided.

# 5 casper8

## 5.1 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.

## 5.2 Vulnerability

In this case the vulnerability is given by the use of the strcpy function allow us to overflow the return address of greetUser, and the loaded library allow to call the system function and launch a shell.

## 5.3 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, I used gdb command `p &system` while running the program using make.

Since I want to pass the string "/bin/xh" that is nowhere inside the library I can put it inside the buffer because even if the stack is non executable I can still use it to store arguments that can be passed to functions. In this case instead of NOP I used spaces to make some padding before and after "/bin/xh".

To make it possible to store spaces I can pass the variable inside bash through "", and 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.

The string in input looks as follows:

1. A string composed by a lot of spaces, "/bin/xh" and some other spaces,
2. `system` function address,
3. `exit` function address,
4. the address of the string.

To find the address of the string I used instead of spaces some A's and B's before and after /bin/xh respectively, in this way the first time I runned the program (using the buffer address) I was able to see where I was pointing (at some B's) and thus change a bit the address until /bin/xh showed up. At this point I put spaces and the attack worked.

## 5.4 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.

Checking for the length of the string in input before copying it into the buffer can also help to mitigate attacks.

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

## 5.5 Advanced level casper81

This level checks for the environmental variables which I did not use in exploiting level8, so the same exploit can be used.

## 5.6 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). Adding four more spaces made the attack work.

## 5.7 Advanced level casper82

This program checks for non-ascii characters inside the buffer, but just for the expected length (666 byte), so the same attack of the other levels still works.

In particular the attack is the same as casper80 which has 682 byte between the buffer and the return address of the function (686 to overwrite it).

# 6 casper10

## 6.1 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.

## 6.2 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

## 6.3 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.

## 6.4 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.