Casper wargame Solutions

F.

December 2018

Contents

1 Overview

Passwords retrieved:

Level	Password	Approximate time
casper4	zssEylQhyfOdX0H7OOKFxEXG0iY9Y7PL	8h
casper40	hcRzYRX24hMUQfGN4comHv8pXgl7rgiK	1h
casper41	qQftts3uK2DjrNxfEPzXTyOpB20vMakY	2h
casper42	1RbjDY7vpb1RvjvLiAN9skfLL3MTZABD	1h
casper5	LzroSg7w54LlHxKZbHVEGYj2UPs1WEUG	30 min
casper50	SuzPtkbto6T8w0H43MQimqOoH7ntIoVh	30 min
casper51	TOv1elxDKtaFgGfWTyyLQ2uzivxI36X4	30 min
casper52	rWhAgxz2U89eRx57LyCmEWunbCW39fAO	30 min
casper6	l1BcWzPZaWMGOM0kaWimJmmT40KmDE9o	5h
casper60	EoWZv5u7pnBkchLK30vXhiyH1sDexpCB	1h
casper61	K14irUGErx7IC3Gvh5ZdO5sIKt5eQCde	2h
casper62	qe3YqqaQSkvJhTEMxt68uUwemCc9d1WI	1h
casper7	OEhxYs88CiZLNXnFs3NV9Tu9IAY4CLhu	30 min
casper70	5qxWx5fVrtK0g21indUP4MHEauEclQ3D	30 min
casper71	YojCanKvrijSB4rYLiG06BZiASkrChK9	30 min
casper72	Yt8DOcwgDID1ie51GBP0SOLjMUHdBrvd	30 min
casper8	AAqtUL09LWefJlJFvTg0SpVg0j89DGe2	6h
casper80	5CS80xbUrxgjbrU3BBS5YslY00qoUd5J	1h
casper81	yMxdglE84E0NpXIXXRcukr8uEWkIHE6w	1h
casper82	MABIQpR4omrMCmdc7PwxbRNm5EPHzYX5	1h
casper9	Tr5upeGMqqat7Yrf6sbhP5TY7fpa95aV	30 min
casper90	s5pSJqTCK5CURGigQOeh6eHgMgLgsqRB	30 min
casper91	cYzVFAGhOWesuPQUgTj9kMpkA7khzK31	30 min
casper92	XmGlzIMRNxlIoAfOKLvnqpEYd7dKOTl3	30 min
casper10	BpI02agB8PmOKDI1LRvMeQJWi2GrzgWA	2h
		TOT: 38 h

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. The important variables are:

- `char buf[666]` inside `greetUser`,
- `char *s` argument of `greetUser` function

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 ran the program, the address found is 0xbffff146. Then, by using `info frame` command I am 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:

- 100 NOP,
- the shellcode (21 byte),
- some padding to fill the buffer and reach the return address (557 byte),
- 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 instructions 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 the attack more robust against small changes in the addresses: the shellcode will

be executed because it is preceded just by NOP instructions. To make sure the overwritten return address will point somewhere between the NOP before the shellcode and not to too low I changed a bit the address of the buffer by putting a slightly higher number, in this way it will be between the NOP's.

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. It's possible to do this by using the `strncpy` instead of `strcpy` which takes three arguments: the destination, the source and the maximum number of characters that can be copied.

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 being 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.

Also the distance between the buffer and the return address is changed, now I need 682 byte to reach it (686 to overwrite it).

So the string provided in input is:

- 682 A's
- 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 addresses. 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 (final address 0xbffff880).

As last thing in the bash script I remove the environmental variable.

2.7 Advanced level casper42

This level checks for non-ascii characters inside the buffer but it search for them just for the buffer expected length that is 666 bytes, it does not check for the entire length of the string provided in input. 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

This level is similar to casper4 but it takes the input from stdin instead of as argument from the terminal. The most important variable is `char buf[666]` inside `greetUser`.

3.2 Vulnerability

This program is vulnerable because it uses `gets` function to read the input, this function does not check for the length of the input, so an attacker can use this vulnerability to overflow the buffer and overwrite the return address of `greetUser`.

3.3 Exploit description

Using gdb I discovered how many characters I need to overwrite the return address of `greetUser` from the buffer, which is 678+4 bytes, and the address of the buffer.

The string that is provided as input is composed as follows:

1. Some padding characters (100 Nop),
2. The shellcode,

3. some other padding to fill the buffer and reach the return address (557 NOP)

4. the address of the buffer.

Keeping the input open (with cat -) makes it possible to use the shell when it's launched.

3.4 Mitigation

To mitigate this attack the length of the input provided by the user should be checked, the function `gets` should be replaced with the safer version `fgets` which has an additional arguments that tells how many characters can by copied.

3.5 Advanced level casper51

The same exploit of the level casper5 works also for this program because in casper51 is added a control on the environmental variables which are not used for the attack.

3.6 Advanced level casper50

To exploit this level, which checks for NOP instruction for the entire length of the input provided I used an environmental variable to store the shellcode (together with some NOP's), filled the buffer with A's and overwrite the return address with the address of the environmental variable.

In this case I also needed to add four bytes in the input string since the distance between the buffer and the return address has changed.

3.7 Advanced level casper52

This level checks non ascii characters for the expected length of the buffer (666 bytes), the same exploit used in casper50 works also for this level because the first 666 characters are A's.

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. The relevant

variables are:

- `somedata` struct which contains:
  - `char buf[666]`
  - `void (*fp)(char *)` function pointer
- `argv[1]` in main which is the input provided by the user

4.2 Vulnerability

This program is vulnerable because it is possible to overwrite the function pointer of the struct, 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:

- some NOP instruction(597 bytes)
- shellcode (21 bytes)
- some more NOP to fill the buffer an reach the function pointer (50 bytes)
- the address of the buffer

Since the address of the buffer can vary a bit, some NOP instructions at the beginning are necessary to make sure that the exploit is robust against small changes. Also pointing to a slightly more higher address makes the attack more robust because it ensures that it's landing inside the buffer and not too low on the stack.

4.4 Mitigation

To mitigate this attack a check of the length of the string copied it's necessary, this can be done by replacing `strcpy` with the safer version `strncpy` which has a length parameter.

Another thing that can help mitigate this attack is using ASLR(Address Space Layout Randomization) which makes more difficult to retrieve the address of the buffer since it will vary at every execution of the program.

Also if the `buf` variable and the function pointer are stored in the reverse order this attack it's not possible because overflowing the buffer would not lead to the function pointer.

4.5 Advanced level casper61

This program adds an additional check on 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 provided as input some 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 of the environmental variable 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:

- 668 A's
- the address of the environmental variable (0xbffff8a0)

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 casper7

5.1 Description

This level is similar to casper6, however it takes input from stdin instead as argument to main.

5.2 Vulnerability

This program is vulnerable because `gets` function does not check for the length of the input, so it's possible to overwrite the function pointer of the struct.

5.3 Exploit description

To exploit this level I first checked for the distance between somedata.buf and somedata.fp, which is 668 bytes. So I filled the buffer with some NOP, the shellcode, some other NOP (in total 668 bytes) and then the address of the buffer.

5.4 Mitigation

To mitigate this attack a check of the length of the string in input it's necessary, this can be done by replacing `gets` with the safer version `fgets` which also has the length parameter.

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

Also if the `buf`variable and the function pointer are stored in the revers order this attack it's not possible because overflowing the buffer would not lead to the function pointer.

5.5 Advanced level casper71

This program cleans the environmental variables, which I did not use to exploit casper7. However to make the attack work I updated the address of the buffer.

5.6 Advanced level casper70

This level adds a check on the input, if there is a NOP it will stop the execution. To exploit this level I created a new environmental variable with

the shellcode (and some NOP) and provided as input 668 A's and the address of the environmental variable.

5.7 Advanced level casper72

This level check if there are some non-ascii characters in the expected length of the buffer(666 bytes), to exploit this level I used the same attack used for level casper70.

6 casper8

6.1 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. The code is the same as casper4. The relevant variable are:

- `char buf[666]` inside `greetUser`,
- `char *s` argument of `greetUser` function

6.2 Vulnerability

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

6.3 Exploit description

To make this exploit work I first looked for how many input characters make the program 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`, in this way I discovered 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 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, in this case the program exits instead of going in segmentation fault.

The string in input looks as follows:

- A string composed by a lot of spaces, "/bin/xh" and some other spaces (total of 678 bytes),
- `system` function address,
- `exit` function address,
- the address of the string.

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

6.4 Mitigation

Checking for the length of the string in input before copying it into the buffer can mitigate this attack, to do this `strncpy` function can be used instead of `strcpy`.

Also 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.

6.5 Advanced level casper81

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

6.6 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 padding spaces made the attack work.

6.7 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).

7 casper9

7.1 Description

This program is similar to casper8 but it takes input from stdin instead from the terminal.

7.2 Vulnerability

This program is vulnerable because it uses `gets` function to get the input from the user, this function does not check for the length of the input.

7.3 Exploit description

To pass this level I used an environmental variable to store the "/bin/xh" string together with some spaces, filled the buffer (and everything between it and the return address) with some A's (678) in order to reach the return address and then the `system` function address followed by `exit` and the address of the environmental variable.

7.4 Mitigation

To make this program no longer vulnerable to this attack is possible to use `fgets` function which checks for the maximum length that can be taken in input.

Also enabling ASLR makes more difficult to retrieve the addresses of the functions since they change at every execution.

7.5 Advanced level casper90

The same approach can be used for this level, but the distance between the buffer and the return address is now 682 bytes.

7.6 Advanced level casper91

To exploit this level (that cleans environmental variables) I decided to use the buffer itself to store the string "/bin/xh", as for casper8.

7.7 Advanced level casper92

The same exploit of casper90 works for casper92 since this program only checks for non ascii characters in the first 666 bytes in input.

8 casper10

8.1 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.

The relevant variables are:

- `char buf[666]` inside `greetUser`,
- `char *s` argument of `greetUser` function

8.2 Vulnerability

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

A format function like `printf` expects as input a format string, that is a string that contains text and format specifiers. If the attacker provides in input format parameters (specified by %) inside a string that is not sanitized, he/she can read and write in memory.

In particular:

- `%n` specifier allow to write the number of characters that have already been formatted to the memory address stored the next place in memory,
- and `%x` read the memory.

So by using `%x` an attacker can reach a place in memory where the address of where he/she wants to write is stored, and then use `%n` to write in that place in memory.

8.3 Exploit description

To make this attack work I have to retrieve the address of `isAdmin` variable and use the format string vulnerability to change it (any value except 0 will work since it's a binary variable).

To do so I need to move the stack pointer to a location where 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, the buffer is stored in a 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 `%x` parameters prints back the A's, at this point I know the number of `%x` parameter I need to reach the beginning of the buffer.

Then 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,
- nine `%x` parameters to reach the beginning of the buffer,
- `%n` to write in the address.

8.4 Mitigation

By adding the format specifier `%s` to the printf in line 9 this attack is no longer possible since the string in input will be read as string and putting specifiers inside it will not have the desired effect.

To make this program more secure it's better enable ASLR (Address Space Layout Randomization,)that will make more difficult to retrieve the address of the variable an attacker wants to change.

9 Notes

1. The shellcode used in all levels that require it is the one suggested in the assignment website, with one byte changed to launch /bin/xh instead of /bin/sh (replacing \x73 with \x78).

2. All attacks can be launched using `make exploit`*n*.