Casper Hacker Wargame: Solutions

Wolfgang Möllmann

December 2019

Level	Password	Time spent
4	tnL7L3hY0DVMqEHtrkySQ2M7XwYUNyjE	10 hours
6	EetVSPHTbn3M1moui3Gg88DbItByyPWz	3 hours
8	q6UYWuXz0iv7b65t1qnHAFBeL4w41YEQ	7 hours
11	BlquVlit6YW2Ks5vLzSml3PGvFqhWX2k	6 hours
40	MRU9gJREyRWO6B5Jia1FQaimA1OMvMNc	2 hours
60	84imr31seHIt4O0vQPqonMRdNUgOHXMW	2 hours
80	swaysxx5L11ycLAOP3urZ1nLfFlocrqY	1 hour
Total		31 hours

1 Casper4 solution

1.1 Description

Casper4 is a program which reads an input from the user, which should be his name, and prints the message "Hello " + input + "!". A buffer of size 999 is used to store the input of the user.

1.2 Vulnerability

This program contains a vulnerability because the strcpy on line 6 does not limit the length of the input the user provides. This allows the user to provide a longer input than fits the size of the buffer, and thus overwrite other data in the stack. It is now possible to overwrite the return address in the stack and let it point to our Shellcode.

1.3 Exploit description

To exploit this vulnerability, we provide an input longer than the input buffer. More specifically, we will provide an input that will overwrite the return address in the stack. We first look for the begin address of the buffer by using gdb and setting a breakpoint just after the function call strcpy. We execute with a valid input and can now inspect the memory address of the buffer. We know that the buffer has a capacity of 999 bytes. Now with some trial and error we will overflow this buffer to guess the number at which the return address can be found. In

this case a total of 1015 bytes will be enough with the last 4 bytes overwriting the return address. Now we just fill this space with 990 times '\x90', which is our NOP and will do nothing. After that we deploy our Shellcode, where '\x73' is replaced by '\x78' to execute '/bin/xh' instead of '/bin/sh'. After that we overwrite the return address by the pointing to the begin address of the buffer that we got earlier. The target computer saves memory addresses in little endian, so we will have to revert our return address in the input. It is good practise to increase the return address and point somewhere in the middle of the buffer to mitigate for the change that your begin address is slightly off. This code will now run through all NOP's into our Shellcode and we have successfully exploited this program.

1.4 Exploit usage

A working exploit is included in the tar-ball and can be launched with make exploit4.

1.5 Mitigation

This vulnerability can be prevented by replacing the vulnerable strcpy with strncpy and as argument the size of the buffer. This will prevent the possibility to overflow the buffer.

2 Casper6 solution

2.1 Description

Casper6 is a program which reads an input from the user, which should be his name, and prints the message "Hello " + input + "!". A Heap is now used to store the input of the user.

2.2 Vulnerability

This program contains a vulnerability because the strcpy on line 21 does not limit the length of the input the user provides. This allows the user to provide a longer input than fits in the maximum size of the heap, and thus overwrite other data. In this case we have a pointer just above the heap in the data_t. It is now possible to overwrite this pointer with an memory address and let it point to our Shellcode.

2.3 Exploit description

To exploit this vulnerability, we provide an input longer than the input buffer. More specifically, we will provide an input that will overwrite the pointer address in the struct. We first look for the begin address of the heap by using gdb and setting a breakpoint just after the function call strcpy. We execute with a valid

input and can now inspect the memory address of the heap by inspecting:p &somedata. We know that the heap has a capacity of 1000 bytes (because 999 is not a multiple of four another zero byte will be added for padding). In this case a total of 1004 bytes will be enough with the last 4 bytes overwriting the pointer address. Now we just fill this space with 979 times '\x90'. After that we deploy our Shellcode. After that we overwrite the return address by the pointing to the address of the beginning of the heap that we got earlier. It is good practise to increase the return address and point somewhere in the middle of the heap to mitigate for the change that your begin address is slightly off. This will run through all NOP's into our Shellcode and we have successfully exploited this program.

2.4 Exploit usage

A working exploit is included in the tar-ball and can be launched with make exploit6.

2.5 Mitigation

This vulnerability can be prevented by replacing the vulnerable strcpy with strncpy and as argument the size of the heap. This will prevent the possibility to overflow the heap.

3 Casper8 solution

3.1 Description

Casper8 is a program which reads an input from the user, which should be his name, and prints the message "Hello " + input + "!". A buffer of size 999 is used to store the input of the user. An important difference to Casper4: the stack is non-executable.

3.2 Vulnerability

This program contains a vulnerability because the strcpy on line 6 does not limit the length of the input the user provides. This allows the user to provide a longer input than fits the size of the buffer, and thus overwrite other data in the stack. Because in this program the stack is non-executable we cannot just overwrite the return address and let it point to our Shellcode like in casper4. We can circumvent this restriction by using a "Return-to-libc" exploit.

3.3 Exploit description

To exploit this vulnerability, we provide an input longer than the input buffer. More specifically, we will provide an input that will overwrite the stack frame. To circumvent the restriction on executing on the stack we will overwrite the

current stack frame with a fake system call with its own stack frame. We begin by overwriting the current return address with an address to a function in a libc library. After that we put a dummy return address for the function to return too after libc function is complete. Our objective is to execute /bin/xh, that is why we call the library function system() for our libc function. This memory address can be easily found by using: p &system. After that we will put the fake return address on stack frame. It is good practise to let this return address point to the library function exit(), otherwise if putting a a return address that is not accessible by the function will result in a segmentation fault after our malicious code has been executed. This could be logged and the IT security team could be informed of this. That's why it is good practise so use the exit() library function to make a clean exit after executing your Shellcode. This can also easily be found with p &exit. Now we need to provide the argument for the system() call. This argument should be a pointer to '/bin/xh'. We can simply put this string after we have given our pointer address and calculate the offset needed to point to the exact memory space. We look up the beginning of the stack and add 1023 to it to let it point to our '/bin/xh'. To make it easier to find the right memory address we can use NOP's. In this case we can use '/' as NOP, because even after inputting multiple slashes the shell will still be executed. This exploit will drop us in the shell and after that will exit cleanly and making our exploit work.

3.4 Exploit usage

A working exploit is included in the tar-ball and can be launched with make exploit8.

3.5 Mitigation

This vulnerability can be prevented by replacing the vulnerable strcpy with strncpy and as argument the size of the buffer. This will prevent the possibility to overflow the buffer.

4 Casper11 solution

4.1 Description

Casper11 is a program which prints the message "Enter your name: ", after that reads an input from the user. It then writes "Hello " + input + ", you have role "+ username.role. If thisUser.role is an admin, then the shell /bin/xh will be executed.

4.2 Vulnerability

This program contains a vulnerability because the gets on line 20 does not limit the length of the input the user provides. This allows the user to provide a

longer input than fits the size of the heap, and thus overwrite other data. In this case we have a pointer just above the heap in the user_t. It is now possible to overwrite this pointer with another memory address.

4.3 Exploit description

The main objective to open the shell is by making our user the admin. It is possible for the user to provide a longer input than fits the size of the heap, and thus overwrite other data. In this case we can overwrite the pointer that points to the role_t struct. We cannot manipulate any data in that structure. In the code we can see that isAdmin needs to be 0x00000001 in order to pass the test. We will use gdb and disas main to see in which block of memory the program executes on. We will now use the find command to look for the specific address that contains 0x00000001. When this address is found we need to keep in mind that the 'rolename' will allocate 32 bytes in the heap. We calculate now the correct return address by subtracting 32 bytes from the found memory address. The check will now pass and the exploit is complete.

4.4 Exploit usage

A working exploit is included in the tar-ball and can be launched with make exploit11.

4.5 Mitigation

This vulnerability can be prevented by replacing the vulnerable gets with fgets and as argument the size of the buffer. This will prevent the possibility to overflow the buffer.

5 Casper40 solution

5.1 Description

Casper40 is very similar to Casper4, with the only exception now that it now will check our input for NOP's (\x90) and if it detects one the program will immediately exit.

5.2 Vulnerability

The main Vulnerability of casper4 is still not closed. It still allows the user to provide a longer input than fits the size of the buffer, and thus overwrite other data in the stack. It is still possible to overwrite the return address in the stack and let it point to our Shellcode.

5.3 Exploit description

The general idea is the same as Casper4. We want to overflow the buffer and overwrite the return address to our Shellcode. We will not fill up the buffer with NOP's, but instead we will use the letter 'A'. We determine the begin address of the buffer. We calculate how much bytes we need to overwrite the return address, in this case this is 1019 bytes. We fill up the input with 994 times 'A', then our Shellcode and then our return address. Because we are not using NOP's we will have to calculate the exact location of our Shellcode. This is simply taking the found begin address of the buffer and increasing it by 994 (the amount of A's). This will run our Shellcode and we have successfully exploited this program.

5.4 Exploit usage

A working exploit is included in the tar-ball and can be launched with make exploit40.

5.5 Mitigation

Casper40 tried to mitigate this Vulnerability by not allowing the use NOP's, but this was not the main problem. It just made it easier to execute our Shellcode by just pointing somewhere in the stack. By calculating the exact start address of our Shellcode we can still exploit this program. This vulnerability can be prevented by replacing the vulnerable strcpy with strncpy and as argument the size of the buffer. This will prevent the possibility to overflow the buffer.

6 Casper60 solution

6.1 Description

Casper60 is very similar to Casper6, with the only exception now that it now will check our input for NOP's (\x90) and if it detects one the program will immediately exit.

6.2 Vulnerability

The main Vulnerability of casper6 is still not closed. It still allows the user to provide a longer input than fits the size of the buffer, and thus overwrite other data in the heap. It is still possible to overwrite the pointer address in the heap and let it point to our Shellcode.

6.3 Exploit description

The general idea is the same as Casper6. We want to overflow the heap and overwrite the pointer address to our Shellcode. We will not fill up the buffer

with NOP's, but instead we will use the letter 'A'. We determine the begin address of the heap by inspecting 'somedata'. We fill up the input with our Shellcode then 979 times 'A' and then our pointer address. Because we are not using NOP's we will have to point to the exact location of our Shellcode. This is simply the found begin address of the buffer. This will now run our Shellcode and we have successfully exploited this program.

6.4 Exploit usage

A working exploit is included in the tar-ball and can be launched with make exploit60.

6.5 Mitigation

Casper60 tried to mitigate this Vulnerability by not allowing the use NOP's, but this was not the main problem. It just made it easier to execute our Shellcode by just pointing somewhere in the heap. By calculating the exact start address of our Shellcode we can still exploit this program. This vulnerability can be prevented by replacing the vulnerable strcpy with strncpy and as argument the size of the heap. This will prevent the possibility to overflow the heap.

7 Casper80 solution

7.1 Description

Casper8 is a program which reads an input from the user, which should be his name, and prints the message "Hello " + input + "!". A buffer of size 999 is used to store the input of the user. An important difference to Casper8 is that now '\x90' are not allowed in our input.

7.2 Vulnerability

This program contains a vulnerability because the strcpy on line 6 does not limit the length of the input the user provides. This allows the user to provide a longer input than fits the size of the buffer, and thus overwrite other data in the stack. Because in this program the stack is non-executable we cannot just overwrite the return address and let it point to our Shellcode. This problem can be circumvented this restriction by using a "Return-to-libc" exploit.

7.3 Exploit description

Because we have not used any '\x90' NOP's in our Casper8 exploit we can apply the same principles in this program. The only difference there is, is that the return address is now up 4 bytes higher in the stack.

7.4 Exploit usage

A working exploit is included in the tar-ball and can be launched with make exploit80.

7.5 Mitigation

This vulnerability can be prevented by replacing the vulnerable strcpy with strncpy and as argument the size of the buffer. This will prevent the possibility to overflow the buffer.