Casper CTF: Solutions

Siebe Dreesen

1 Overview

Level	Password	Time spent
3	mlHvsSM2qDSqIK1bOyixamw7O7igKdBF	10 minutes
4	HikOeJRoG2kYCI1QrHBf0bouflFWStPy	4 hours
5	XRTOmu0ToSDrWsPMv7UFPqRcDJFUVf6E	8 hours
6	vb56zXrIsjRPrpM81f5EsqnAjBKYnft8	5 hours
61	8NuwK9iWoCpNlmLNVKmagieWe0j0StHl	1 hour 30 minutes
62	GG2RieHeoogpMJF1qFyQhfb8Vm7SUh6P	20 minutes
63	oADxWU6GKYmesTfKRiihFr3wK8bMnFxM	10 minutes
7	UlzsukgATPSjI4eiMU8JkmAi3LKBvLMP	3 hours
8	42PX6QWbUgN3EeEWrBkI1KJZSip6yCQV	2 hours
Total		24 hours 10 minutes

2 Casper 4 solution

2.1 Description

This program takes 1 argument as input, if more are given it will print the first argument and stop the program immediately. The program contains a struct which contains a buffer of 775 characters and a function pointer. When less then 2 arguments are given the first will be copied into the buffer of the struct using strcpy() and the function pointer will be called. Normally this function pointer will point to the greetUser function which will print given string.

2.2 Vulnerability

This program contains a memory management vulnerability, specifically a buffer overflow vulnerability where we can overwrite a function pointer. The struct contains a buffer and a function pointer which will be allocated underneath the buffer in memory. The function strcpy() which is unsafe will copy the first argument but does not do bounds checking. This means the buffer can be overflowed and the function pointer can be overwritten by giving an argument that is bigger than the allocated space for the buffer of 775 bytes.

2.3 Exploit description

The vulnerability can be exploited by giving a well-chosen argument that is bigger than the buffer space and overwrites the function pointer. The argument contains shell code which will spawn the /bin/xh shell.

The specific argument is build from the following bytes:

- A NOP sled of 755 byte
- The shell code of 21 bytes
- Address in the sled which will be pointed to the shell code and overwrites the function pointer

This argument will be copied to the buffer and overwrite the function pointer with the right address. By using a NOP sled the exploit becomes more robust because any address in the sled can be called to execute the shell code. Note that to overflow the buffer actually 776 bytes are used because the buffer is allocated in memory as words which means 194 words = 776 bytes.

2.4 Mitigation

A possible mitigation would be to use strncpy() instead of srtcpy() which will only copy N characters and the buffer will never overflow. Other possible mitigations are non-executable data so that the shell code cannot be executed or address space layout randomisation which will make it harder to point to the right address of the shell code. Stack canaries would not work because they are inserted between the return address and the local variables, in our case only the local variables are changed.

3 Casper 5 solution

3.1 Description

This program again takes 1 input which is expected to be a username. The program copies this name to the dedicated buffer using the unsafe strcpy(). The program stores users in a struct containing a name and a pointer to a role which contains the name of the role and the authority level. If the user has authority level 1 the /bin/xh shell will run. The default role has authority 0.

3.2 Vulnerability

The vulnerability is again because of the unsafe strcpy() which let's the attacker possibly overwrite the role pointer to a role struct with authority 1 by overflowing the buffer. This will cause the shell to automatically execute.

3.3 Exploit description

I have chosen for a data-only attack by corrupting the role pointer. I did this by giving an argument which had 776 chars and 1 address (4 bytes). The random chars overflowed the allocated space for name causing the role pointer to be overwritten by the chosen address.

This problem took longer than expected because I first tried to exploit this by letting this point to a custom role struct in the name part and letting the pointer point to that struct instead of the default. This did not work because you cannot give an argument with null bytes which are necessary to represent integer 1.

So the solution was to find a couple of bytes of the right length in the memory that already contained int 1 to represent the struct. I did this by searching in the c library. I was able to find an address which had 32 random bytes and then 4 bytes which represented 1. I gave this address as argument and the shell executed.

3.4 Mitigation

The easiest defence would again be to not use strcpy() but strncpy() instead, this would make sure the attacker cannot overflow the name by a given argument. Another mitigation would be to use address space layout randomisation which would make it very hard for the attacker to find the address of certain bytes that would have the role structure. Because it is a data-only attack non-executable data would not work as a mitigation.

4 Casper 6 solution

4.1 Description

In this program there again is a buffer of 775 bytes (actually 776 in memory) which will be filled with a given name by a user. The user can give input when the program executes gets() which reads characters of the input stream and adds a 0x00 byte at the end, this function is unsafe because the input can be as big as desired.

4.2 Vulnerability

The vulnerability is caused by the gets() function which does not do bounds checking. The attacker can overflow the buffer and overwrite the return address to the desired shell code. This is called call stack smashing.

4.3 Exploit description

This vulnerability can be exploited by given a well-chosen argument as input which contains shell code and overwrites the return address to the self written shell code. The shell code in this exploit will spawn the //bin/xh shell. A NOP sled will be used to add robustness, the return address wil be pointing to an address in this sled.

The input:

– NOP sled (0x90) of 755 bytes
– Shell code of 21 bytes
– 11 characters which fill up the space between the buffer and the return address
– The desired return address

Note that is necessary to insert characters between the return address and the shell code because the memory dedicated for the frame pointer cannot be executed.

4.4 Mitigation

A possible mitigation for this would be stack canaries which would warn the system that the return address is overwritten. These are random words added between the local variables and the return address and are checked on changes when returning. Another possibility would be to enable non-executable data so that the shell code cannot be executed but this could be solved by using a code reuse/data-only attack instead. An easy solution would be to use a different function to read the input like fgets() which stops reading when the buffer is full. Also address space layout randomisation would again make it more difficult for the attacker.

4.5 Advanced levels

Casper 61 This advanced level checks wether the given input contains NOP's, it is very hard to find the exact address and if addresses would change it could break the exploit. This is why I changed my approach and instead of giving the shell code as input I added it to the environment variable with 500 NOP's for robustness. The I overflowed the buffer with input just as in the original case but now only chars and an address in the NOP sled to overwrite the return address. The program will iterate over the buffer until the return address and not find any NOP's which means that the protection is bypassed and the shell code will be executed.

I added env -i with some GDB variables to make sure the exploit does not break.

Casper 62 In this level the program is protected by clearing the environment so that the attacker cannot use environment variables in their exploits, so an exploit as in Casper 61 would not work anymore. I was able to solve this by using the same exploit as in Casper 6 solution, the original level which does not use environment variables.

Casper 63 This level is protected by checking if the buffer contains any non ascii characters, this means the shell code cannot be placed in the buffer. The solution from Casper 61 uses an environment variable to solve this and fills the buffer with char A. I reused that exploited which worked, the program only is able to find ascii characters in the buffer and uses the return address to the environment variable which contains the shell code.

I added env -i with some GDB variables to make sure the exploit does not break.

5 Casper 7 solution

5.1 Description

The program is basically the same as the program in Casper 6 solution, except for the fact that non-executable stack is enabled. This means that we cannot make use of self written shell code on the stack.

5.2 Vulnerability

The vulnerability in this level is again caused because of the unsafe gets() function which does not do bounds checking. This means we can overflow the buffer to overwrite the return address and other information.

5.3 Exploit description

Because the stack is non-executable I used a code reuse attack, specifically return-to-libc. The idea is to let the return address point to a desired function from the C library which will be executed on return. The input has to overflow the buffer, set the return address to a function, overwrite the saved return address and add the necessary arguments. I used the system() function with argument "/bin/xh" to execute the shell. The argument will be saved as an environment variable with spaces instead of a NOP sled.

I used the following input:

- 787 characters to overwrite the buffer
- The address of system() (4 bytes)
- The address of exit() (4 bytes) makes sure that it cleanly terminates

– An address in the environment variable which contains /bin/xh, there is padding because of the spaces

After a test on a different day, the exploit was broken. This is because when environment variables are added it changes the address space slightly so I added env -i with some gdb variables to make sure the address does not change too much.

5.4 Mitigation

The simplest mitigation is to use a safe function to get input like fgets() which will make sure the buffer cannot be overflown. There are other defence mechanisms like address space layout randomisation which would make it very difficult for the attacker to find the right addresses or stack canaries which would warn the process that the buffer is overflown. But both only make it more difficult for the attacker but do not give any certainty.

6 Casper 8 solution

6.1 Description

In this level the program calls a function greetUser() which writes a formatted string to a buffer and prints this buffer. The formatted string that will be written to the buffer is the first argument given by the user. If the global variable isAdmin is not 0 the shell is executed, the default value is 0.

6.2 Vulnerability

This is a formatted-string vulnerability which can be exploited by a data only attack because stack canaries and non-executable stack is enabled.

6.3 Exploit description

The exploit is done by changing the isAdmin variable to an integer using the formatted string vulnerability. This can be done by giving an argument with the address of the isAdmin variable which overwrites the variable with the length of the formatted string, the exact value does not matter as long as it is not zero. I found the offset between the given address and stack pointer by giving a formatted string containing "%08x" which will print 4 bytes.

The arguments is built from:

– A random char (1 byte), this is necessary because the address has to be removed a fourfold of bytes from the SP for the attack to work.
– The address of isAdmin (4 bytes)
– 10 times "%08x", 10 is the offset from the address
– "%n" which will print the length of the string to the given address

The offset is caused by the characters already written in the source code of the problem in the sprintf() function.

6.4 Mitigation

This vulnerability can be mitigated by using address space layout randomisation which would make it difficult for the attacker to find the right address of isAdmin. Another way to prevent an attack would be to only use printf() instead of writing it to a buffer so the formatted string cannot be interpreted differently.