Development of Secure Software: Project report

Danilo Peeters

December 2020

1 Summary

Table 1 shows a summary of the levels, time spent and their passwords.

Name	Password	Time
Casper040	`KoWTY9glAmLEzggtpof4tBAS1dDC5h64`	5 hours
Casper060	`4CsId751r3cp67Pa3Apo9B9YNkSvxTSe`	6 hours
Casper080	`yHemERaJlCx0qaitkBIEEkEyijdH4BU6`	2 hours
Casper100	`p4zxxve14nJFigSHWJxN0mOcBbbCvM6n`	2 hours
Casper120	`v35UZPnKCC0xrfvfWrtLqQWajK81d7qw`	4 hours

Table 1: Summary of exploits

2 Casper040

2.1 Description

This program requires the user to pass an argument to the program upon execution. It simply prints "Hello `input`!". The most important variable in this program is the `somedata` struct. This struct contains a buffer of length 987 and a pointer to a function which it will execute. In normal circumstances, this pointer should point to the function responsible for printing.

2.2 Vulnerability

The `strcpy` function on line 22 does not check the size of the variable it has to copy. This allows for overflowing the buffer and writing other code at the addresses which lie beneath the address for the buffer.

2.3 Exploit description

The first part of the exploit consists of checking where the `somedata` struct stores its `buf` and `fp` fields in memory. This is done by using `gdb` and setting

a break point at line 22. The following commands in `gdb` show the memory addresses:

```
(gdb) p &somedata.buf
(gdb) p &somedata.fp
```

The start of the buffer is located at address `0x08049840` and the function pointer's address at `0x08049c1c`. The distance between these addresses is 988 bytes, one more than the size of the buffer.

The payload consists of the following parts:

- A series of NOP commands
- The shellcode executing `/bin/xh`
- Overflow data
- The return address, which is the start of the buffer

The first three parts of the payload should have a length of 988 bytes, such that the return address is the next part to be overwritten.

2.4 Mitigation

This exploit can be avoided by using `strncpy` instead, which requires the programmer to specify a length of data that should be copied.

3 Casper060

3.1 Description

This program takes an input name, and writes it to stdout. More specifically, it contains a struct `User` which contains a char buffer of length 987 and a pointer to a `role_t` struct. This struct contains a 32 char buffer and integer to check whether the user is an admin or not. In normal circumstances, this integer is set to 0 and the admin-code cannot be ran.

3.2 Vulnerability

The vulnerability is found in the `strcpy` function. This function allows for overwriting addressed outside of the buffer range. Stack canaries were implemented for this exercise, so overwriting the program counter is impossible without detection. Instead, the pointer to the `role_t` struct can be overwritten.

3.3 Exploit description

First, the actual structure of the `role_t` struct was checked using gdb. As expected, it contains 32 bytes for the char array and 4 bytes for the integer. The idea of this exploit is to point to a 'fake' struct on address A, such that for bytes A+33 through A+36, an integer value of 1 rests in memory. This way, the if-condition for checking the authority level is met. The first attempt included writing this fake buffer in the space for the name in the `user` struct, although this presented difficulties due to the necessity of passing
`x00` bytes. The second attempt consisted of browsing through memory addresses to find an address B containing an integer 1. The new pointer address is thus given by B-32. The payload thus consists of two elements:

- Junk data to fill up the name buffer;
- The B-32 address, pointing to the fake struct.

3.4 Mitigation

As for casper040, strncopy can be used to prevent overflows.

4 Casper080

4.1 Description

This program takes an input argument, copies it to a buffer and displays "hello `input`".

4.2 Vulnerability

Stack canaries and non-executable stack were disabled, which makes it possible to overwrite the program counter to point to some data that was introduced by the input. Overwriting is possible due to the usage of the unsafe `strcpy` function.

4.3 Exploit

This exploit introduces shell code into the buffer and overwrites the program counter to execute these instructions. Concretely, the address of the program counter was found by breaking on any line in the `greetUser()` function, and checking the saved return address. This is done in gdb using `info frame`. The address of the start of the buffer was also found using gdb using `p &buf`. A quick calculation reveals a distance of 999 bytes between these addresses. Overwriting the next 4 bytes with an address should thus execute our instructions once the program exits the `greetUser()` function. The payload consists of the following elements:

- A series of NOP instructions, as a countermeasure to small address changes;
- The shellcode;
- Junk data filling up the leftover space until the program counter;
- A new address, pointing somewhere in the middle of the NOP instruction.

4.4 Mitigation

The introduction of stack canaries prevents overwriting the program counter without detection. On the other hand, using `strncpy` and fixing the length of the buffer also prevents the user from overwriting other memory addresses.

5 Casper100

5.1 Description

This program takes a command line argument, stores it into a buffer and prints its contents.

5.2 Vulnerability

Stack canaries and non-executable stacks were enabled, so simply injecting shellcode is not possible. It is however possible to once again overwrite into other memory due to the `strcpy` function.

5.3 Exploit

A return-to-libc attack was chosen to exploit this program. An environment variable `XHSHELL` was created with as value 400 spaces and the string `/bin/xh`. The spaces were added to consume memory to account for small changes in memory addresses between execution.

The system address was found by breaking on the first line of the C program and checking the program counter using `info frame`.

The address A_{env} for the environment variable was found by checking the contents of the memory addresses using the following gdb command. To allow for small possible memory changes, the effective address was chosen as A_{env}+200

```
(gdb) x/20s *((char **)environ)
```

The final payload consists of:

- Random data filling up the buffer;
- Random data filling up the space between the buffer and program counter;
- The system address;

- The system return address (not really needed, so 4 random bytes);
- The address of the environment variable.

5.4 Mitigation

Address space layout randomization (ASLR) makes this type of attack very difficult to execute. In this case, using a safe copy function should also prevent buffer overflow which makes this attack possible.

6 Casper120

6.1 Description

This program requires a name argument and prints "Hello `input`". If the `isAdmin` variable is set, the program runs a shell.

6.2 Vulnerability

The unsafe usage of the `sprintf()` function introduces a format string vulnerability. By cleverly passing a specifically constructed string, memory can be read and written.

6.3 Exploit

The first part of this exploit consists of first retrieving the address ADDR of the `isAdmin` variable. This is done as before using gdb. To overwrite this address, a stack dump is requested by inputting the following string:

```
ADDR-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-(...)
```

The amount of `%x` operators is chosen incrementally until the ADDR is found on the stack. After inspection, it seemed that the first byte of the address was written on another address, so an offset byte of
`x90` was added. Finally, a `%n` string was added to write to the memory address.

6.4 Mitigation

A safe usage of the printf function prevents this type of attack (eg. not directly printing the user's input).