Defensive System Programming - 20937

Maman 12

Yehonatan Simian
206584021

Question 1 (20%)

```
bool is_entitled_for_promotional_gift(int ID)
{
    unsigned int bound = 750;
    int credit = get_credit(ID);
    return (credit >= bound);
}
```

Section 1

The weakness in the code above is the comparison between an `int` to an `unsigned` variables. Specifically, the compiler gonna perform an implicit conversion of `credit` from `int` to `unsigned` type, so he can be able to compare these two values.

By definition, the difference between the bit representation of `int` and `unsigned` is in the most-significant bit; in the former it's a part of the number itself, whilst in the latter it's an indication of a negative number. For example, the binary sequence `101` is `5` when representing an `unsigned`, but is `2` when representing `int`.

One way to attack the system would be having a low negative amount of credit in the user's account. That way, the compared credit amount would be very big, passing the value of `750` for every known type of unsigned integer (`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`).

Section 2

One possible fix to the code would be removing the `unsigned` from the declaration of `bound`. The solution looks likes this:

```
bool is_entitled_for_promotional_gift(int ID)
{
    int bound = 750;
    int credit = get_credit(ID);
    return (credit >= bound);
}
```

Section 3

The weakness is fully described in the risk analysis table below:

Question 1 continued on next page. . .

Threat	Unauthorized clients (credit ≤ 750) will be able to receive an unintended promotional gift.
Affected component	Any component that uses the module.
Module details	`is_entitled_for_promotional_gift`.
Vulnerability class	CWE-195: Signed to Unsigned Conversion Error.
Description	The comparison between an `int` to an `unsigned` variables will cause an implicit conversion from `int` to `unsigned` type, which can be used to attack the system by customers with negative credit in order to receive a promotional gift.
Result	Customers with negative credit are able to receive a promotional gift.
Prerequisites	A customer with a negative credit (which is allowed by the company's policy).
Business impact	"Kashe" will lose some precious and valuable dinerosssss.
Proposed remediation	Change `bound` declaration from `unsigned` to `int`.
Risk	Damage potential: 10 Reproducibility: 10 Exploitability: 10 Affected users: 10 Discoverability: 10 Overall: MASS DESTRUCTION.

Table 1: Risk Analysis Table for Question 1

Question 2 (80%)

Section 1

I have compiled the given code on a `Ubuntu 20.04.5 LTS` WSL using the following command: `g++ -m32 -g3 -std=c++17 -o mmn12 mmn12-q2.cpp`. Next, I saw in the code that the environment variable ECHOUTIL_OPT_ON must be set, so I set it using the command `export ECHOUTIL_OPT_ON=true`. Finally, the code could be run using `./mmn12` with any relevant arguments (e.g. `--help` and `--version`).

Section 2

Let's figure what arguments we should give the program in order to call `Handler::unreachable`. First, the handler is only used in the `handle_escape` function. In order to call this function, according to line 180: `if (do_escape && s[0] == '\\')`, we see that two conditions must be met in order for this function to be called. Let's meet these conditions:

1. `do_escape`: on line 161 we see that this variable is set to `true` if and only if the `-e` argument in been sent to the program. Easy peasy.
2. `s[0] == '\\'`: in the previous loop we saw that `argc` is being decremented, and `argv` is being incremented. This means that the condition we're trying to meet is relevant to the next argument; thus, all we must do is begin the argument with two backslashes. Lemon squeezy.

So far, we know that in order to call `handle_escape`, the program calling command must start with: `./mmn12 -e \\`. Now's the interesting part: the function `handle_escape` creates a struct with a buffer string of size 16 followed by a `Handler` object. The function then copies the argument `str`, which is the programs next argument after `-e`, to the buffer. **Here lies the weakness - no one checks that the string we pass actually fits the buffer!** In other words, we can create a buffer overflow and override some of the `Handle` instance in the memory. Specifically, we'd like to override the virtual table, so we can call the function `Handle::unreachable` when it was not meant to be called.

So how do we do that? Easy. On line 86 we see that the function `Handler::interpret` is being called if the first character in the buffer is `x`. No problem. So we actually want to use ~~gdb~~ **gef** to put a breakpoint on the call to `Handler::interpret`. At this point, we know for sure that `Handler` was being constructed, and so it's virtual table. We'll look for the address of `Handler::unreachable` using the debugger, and override the virtual table utilizing our understanding of buffer overflow. As the young people say... Piece'O'cake.

So, to be more precise, the process is:

1. Run the program using the following command: `gdb --args mmn12 -e \\x`
2. Place a breakpoint at the desired function: `b Handler::interpret(char const*)`
3. Run the program: `r`.

- * At the breakpoint, we get a view of the registers' values and (thanks to the usage of gef) the functions they're pointing at. Specifically, we can observe something interesting going on $eax: `$eax : 0xffffc988 → 0x56559e24 → 0x56556ae2 → <Handler::unreachable()+0> endbr32`. This means that the $eax register points to the virtual table (located at `0x56559e24`), which in turn points to the function `Handler::unreachable` (located at `0x56556ae2`).

- ** Now, by the structure of `Handler`, we know that in the virtual table, `Handler::unreachable` come right before `Handler::helper` function pointer. Specifically, 4 bytes before (because we're at 32bit mode). We also know that `Handler` has no member variables, and the only non-virtual member function is `Handler::interpret`, which only calls `Handler::helper`. Hence, the address we want to write using the buffer overflow is `0x56559e24 - 4 = 0x56559e20`.

4. Using our new knowledge, we shall run the program using 15 allegedly meaningless characters after the `\\x` in the second argument, and then the address of the virtual table in little endian: `r -e \\x$(echo -e "Yoni-HaMelech!!\x20\x9e\x55\x56")`.

After the last step we can see that `Cowabunga!` is being printed to the screen, i.e. the system has been successfully hacked and we can rest on our laurels, drinking Oprah Winfrey's polyjuice potion while playing `"Cut the ROP"` on our mobile phones *uwu*.

Section 3

Let's fix the code to handle this vulnerability. As we've seen on Section 2, the weakness appears on the `handle_escape` function, when copying the given string to the buffer:

```
while (*s)
    *p++ = *s++;
```

We'll fix that by copying only the first 15 characters of the given argument string:

```
for (size_t i{}; i < 15 && *s; ++i)
    *p++ = *s++;
*p = '\0';
```

Note: if we wanted to be serious, we'd set a `static constexpr size_t BUFFER_SIZE = 16` as a public static member variable of `Handler`, and then use it in the code. But we ain't serious. We're just cutsie putsie little wannabe shlutzies, ain't we? ^^

Section 4

The weakness is fully described in the risk analysis table below:

Question 2 continued on next page. . .

Threat	Shredder may discover the secret password.
Affected component	`Handler`?
Module details	`Handler::unreachable`?
Vulnerability class	CWE-121: Stack-based Buffer Overflow.
Description	Unguarded user input-length may cause a buffer overflow which can be exploit by the wrong hands to discover the secret password, pass it to Shredder, and risk everyone with world domination where Krembo is illegal and everyone must drive on the left side of the road.
Result	Unwanted execution of `Handler::unreachable` and basically any other ROP attack.
Prerequisites	Attacker must have access to source code in order to understand how to compile and use the debugger and environment variables to exploit the vulnerability.
Proposed remediation	Validate the user input before using it.
Risk	Damage potential: 1 Reproducibility: 1 Exploitability: 2 Affected users: 1 Discoverability: 69 Overall: Nah.

Table 2: Risk Analysis Table for Question 2

Figure 1: Meme