Buffer Overflow Attacks: **Privilege Escalation**

OSES Team Project A.Y. 2022/23 - **Group 03**

Lorenzo Ruotolo *s323107*
Alessandro Vargiu *s314294*
Marco Chiarle *s314730*
Giovanni Santangelo *s308882*

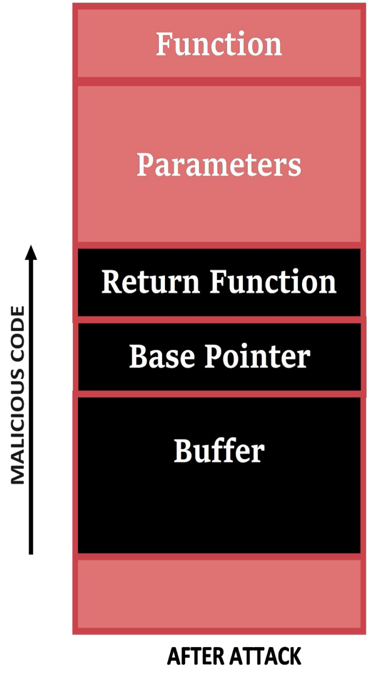
What is a Buffer Overflow?

Definition

A Buffer Overflow is a vulnerability in which a program that writes into its buffer, overruns its boundaries and overwrites adjacent locations of memory.

Focus:

We will focus on stack-based buffer overflows with privilege escalation.

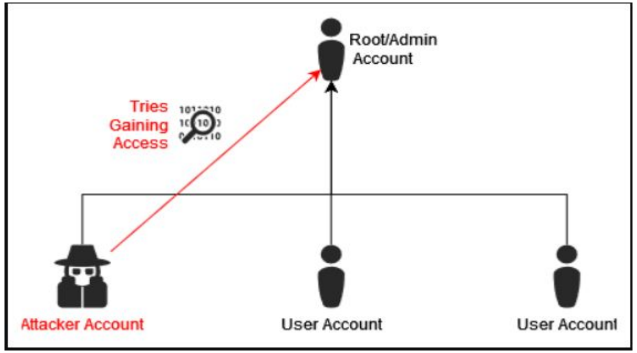


Privilege Escalation

Privilege escalation is a type of attack designed to gain unauthorized access into a system.

Vertical privilege escalation: the attack exploits a lower privilege application to access higher-privilege ones.

Horizontal privilege escalation: lower privilege application or user accesses other data from same-privilege memory locations or users.



Exploitation

The standard exploit of a buffer overflow consists in overwriting the location of the stack which contains the function return address.

The exploit usually targets a vulnerable function (ex. strcpy or gets) that takes data in input without checking its length or boundaries.

When this happens, an attacker can inject executable code into the running program and take control of the system.

Causes

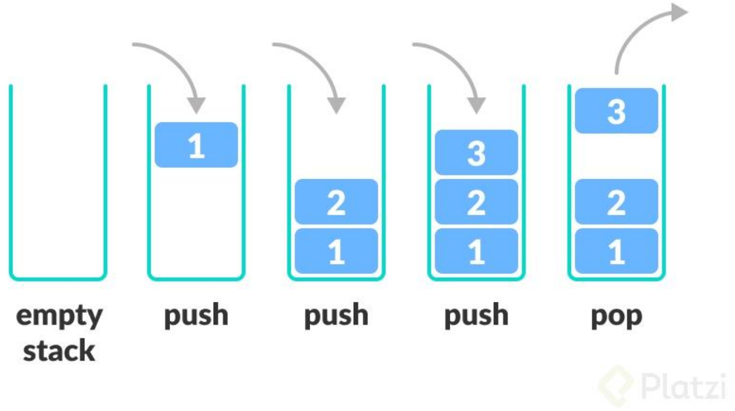
There are different possibilities:

1. **Negligent Input Validation**: Buffer Overflows often result from not sufficient checks in input length or boundaries.
2. **Vulnerable function:** Functions that lack proper input validation like strcpy or gets.

Buffer Overflow in-depth

The Call Stack

- The call stack is the data structure that stores information about subroutines of an executing computer program. It also contains the values for the local variables and the return address for the subroutine.
- Its properties are based on the Stack data structure, which means that the order in which elements are stored operates in what is called **LIFO**.



Stack-based Buffer Overflow

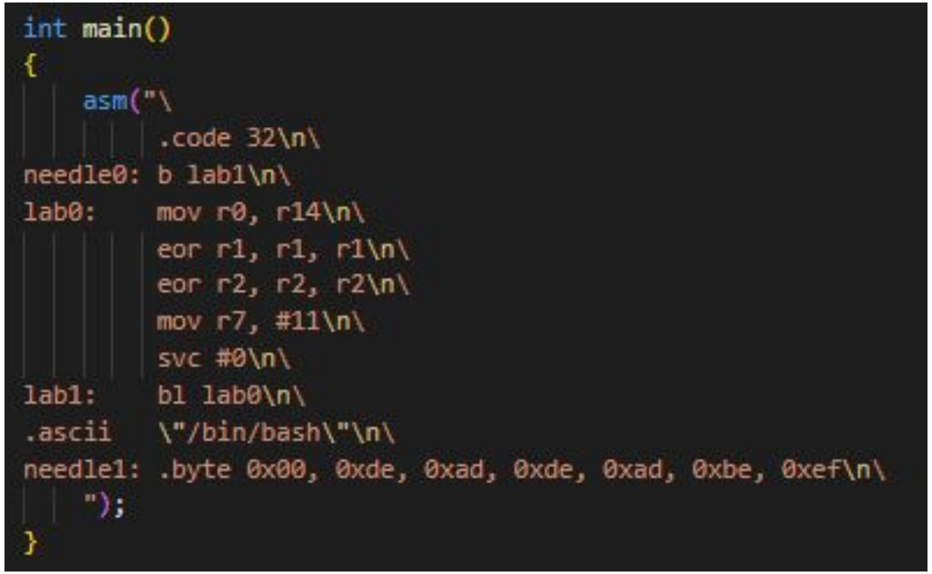
This type of buffer overflow focuses on what is called a **buffer**.

The **buffer** is a part of the stack in which content of local variables are stored.

The key point is to write more data than the buffer can hold, which causes the overwrite of adjacent memory, including the return address memory space, which is usually very close to the buffer.

The Exploit

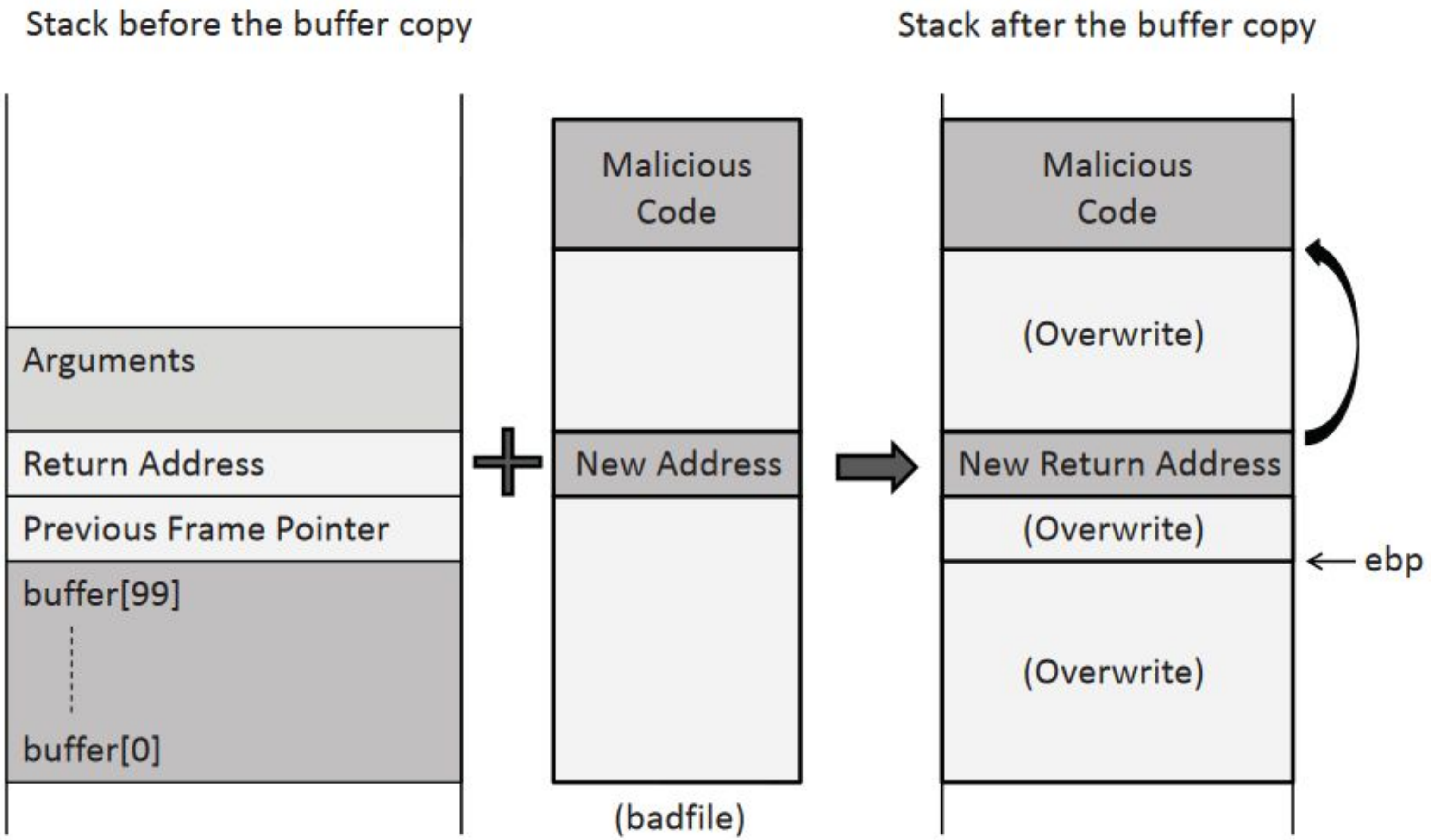
- The objective is to overwrite the memory of the return address with a location pointing on malicious code.
- The best practical way to do this is to store the malicious code (**shellcode** in this case) somewhere inside the buffer.



```
int main()
{
    asm("\
            .code 32\n\
needle0: b lab1\n\
lab0:    mov r0, r14\n\
         eor r1, r1, r1\n\
         eor r2, r2, r2\n\
         mov r7, #11\n\
         svc #0\n\
lab1:    bl lab0\n\
.ascii   \"/bin/bash\"\n\
needle1: .byte 0x00, 0xde, 0xad, 0xde, 0xad, 0xbe, 0xef\n\
    ");
}
```

The Exploit

- Given the nature of memory spaces and its volatility, pointing directly at the start of the shellcode is unreliable.
- If the buffer is filled with ***NOP*** instructions, we can overwrite the return address with the start of the buffer and the sequence of ***NOP*** instructions will act as a slide that sends the program execution to the **shellcode**.
- Once the shellcode is executed we can gain control of the system, for example, executing code which spawns a root shell in our terminal.


Stack before the buffer copy
Stack after the buffer copy
Arguments
Return Address
Previous Frame Pointer
buffer[99]
buffer[0]
Malicious Code
New Address
(badfile)
Malicious Code
(Overwrite)
New Return Address
(Overwrite)
ebp
(Overwrite)

Protection Techniques

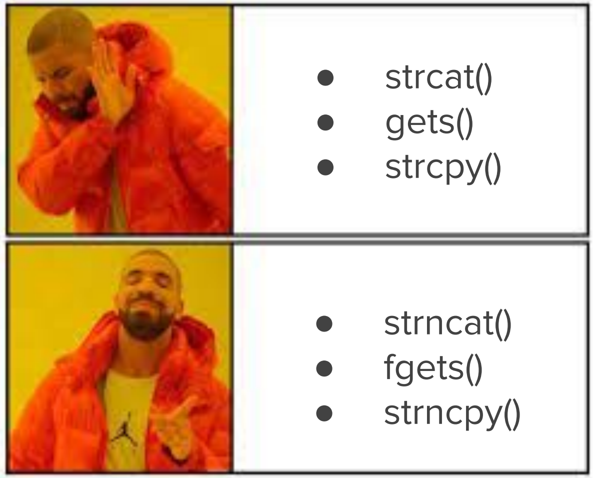
Protection

Operating System Level

1. **DEP**: Data Execution Prevention. Non Executable Stack
2. **SSP**: Stack Smash Protector
3. **ASLR**: Address Space Layout Randomization

Application Level

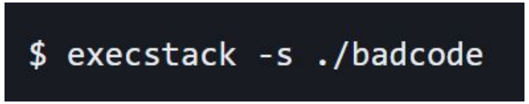
1. Bound Checking Functions.



Our focus is on the Operating System Level.

Data Execution Prevention (DEP)

Protection which forms the **Non-Executable stack**.
DEP adds a layer of protection against common buffer overflows.



```
$ execstack -s ./badcode
```

DEP implementation:

1. hardware (through processor features)
2. software (through operating systems) to provide an additional layer of security

While DEP protects from stack-based overflows, other types of buffer overflows attack can be performed which exploit unprotected regions, like the heap.
(see return to libc which bypasses the non-executebit constraint in DEP)

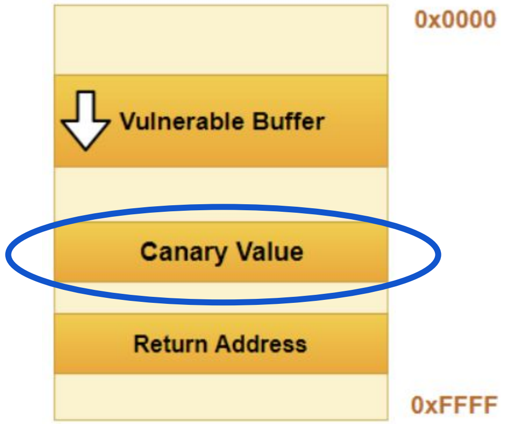
Stack Smash Protector (SSP) (Canaries)

Stack Smash Protector (Canaries): a canary is a value placed close to a buffer. In case of a buffer overflow, the canary value can be the first to be corrupted. This indicates the presence of the attack to the system.

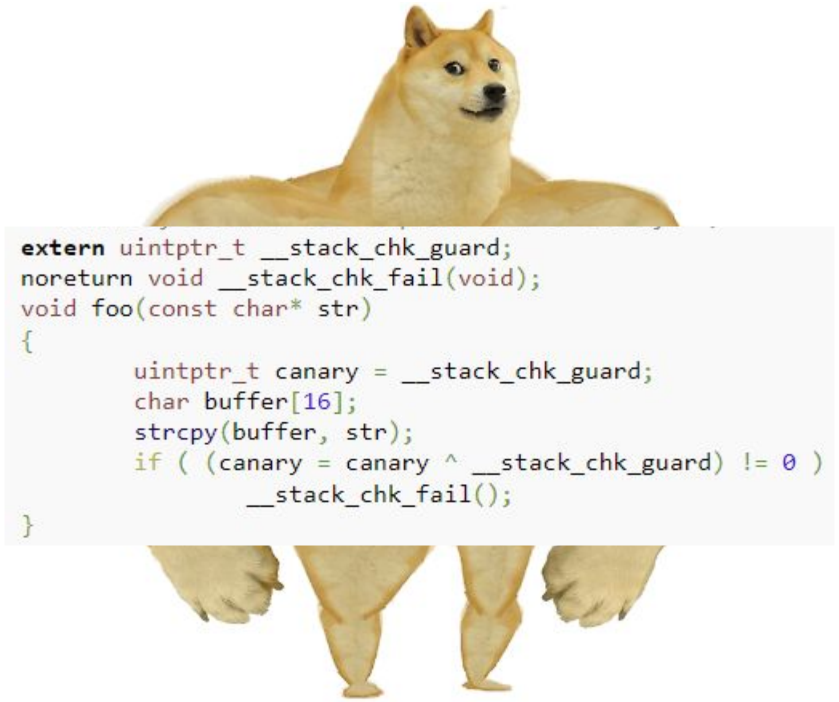
Compilers implement this feature by:

1. Selecting appropriate functions
2. Storing the stack canary during the function prologue
3. Checking the value in the epilogue
4. Invoking a failure handler if canary was “killed”

Some compilers randomize the order of stack variables and randomize the stack frame layout

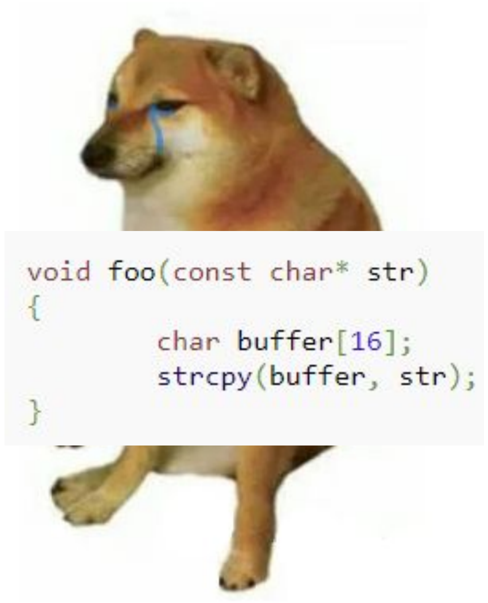


SSP Confrontation Example



```
extern uintptr_t __stack_chk_guard;
noreturn void __stack_chk_fail(void);
void foo(const char* str)
{
        uintptr_t canary = __stack_chk_guard;
        char buffer[16];
        strcpy(buffer, str);
        if ( (canary = canary ^ __stack_chk_guard) != 0 )
                __stack_chk_fail();
}
```

compilation with SSP



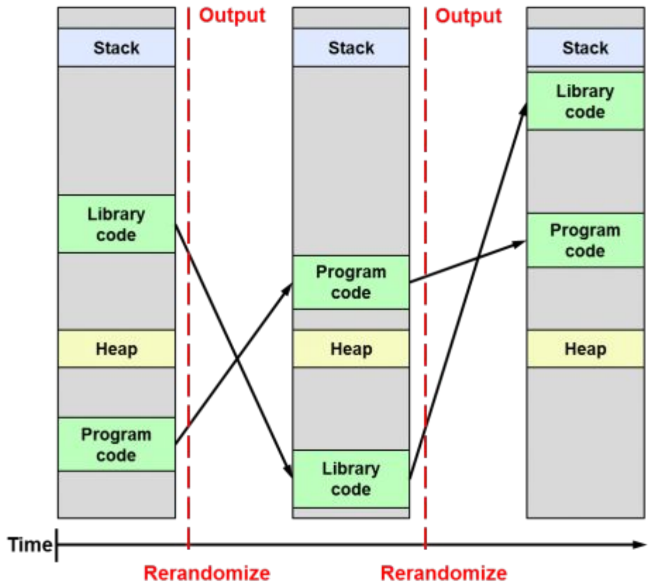
```
void foo(const char* str)
{
        char buffer[16];
        strcpy(buffer, str);
}
```

compilation without SSP

ASLR - Address Space Layout Randomization

ASLR (Address Space Layout Randomization): address spaces of programs are randomly arranged to prevent the attacker to reliably jump to desired locations of memories.

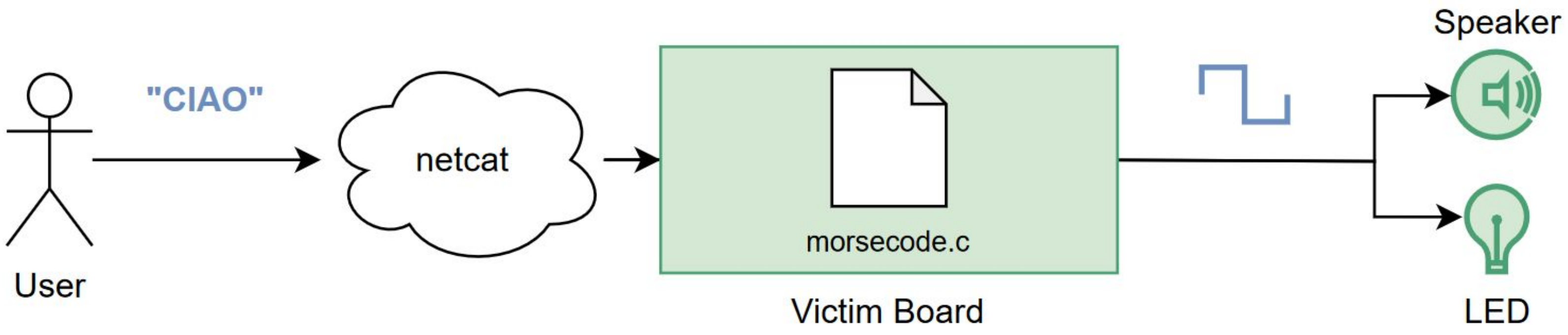
This makes the memory layout unpredictable for attacker programs since it do not know where the next instruction lies.



Time for a Demo

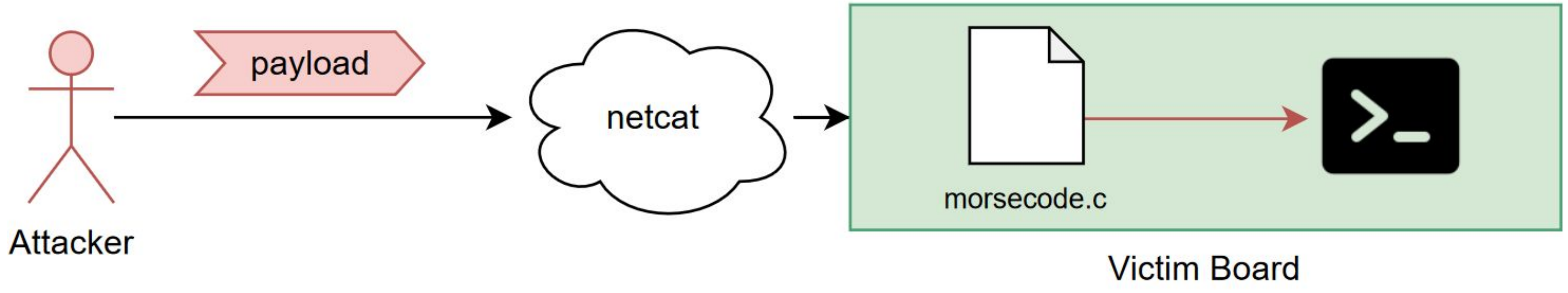
Scenario: the Victim

The **victim** is running a function *morsecode* on its device, a Raspberry board running Linux. The function is exposed to the local network through **netcat**.



Scenario: the Attacker

Inside *morsecode*, the vulnerable **gets** function is used to parse the textual input, and will be exploited by the **attacker** to launch the attack.



Scenario: the Attacker

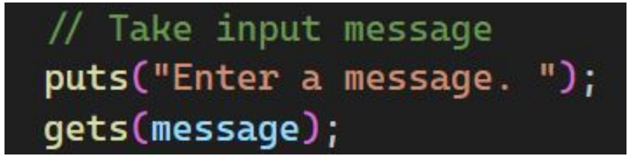
The purpose of the **attacker** is to exploit the vulnerability to get unauthorized control of the board. Before doing that:

1. The attacker gains access to the source code of *morsecode*.
2. Using **gdb** he discovers the address of the buffer to use as return address.
3. Using a the *makepayload* script, the **payload** is prepared utilizing the previously found return address and the binary code for spawning a shell.

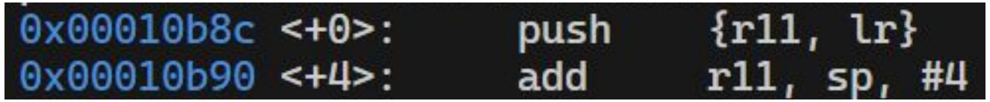
How to find the Offset with GDB

The **offset** is needed to let the script overwrite with byte-precision the return address, starting from the base buffer address.

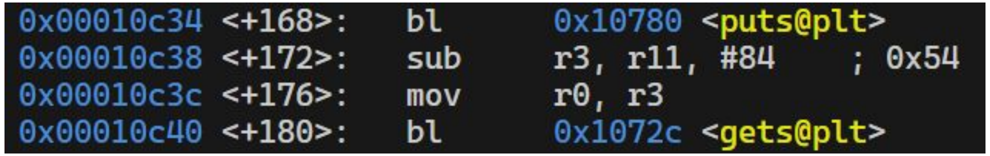
As we can see from the **disassembled** main function, the buffer sits 84 bytes below the *$lr* (Link Register) value, containing the return address value that needs to be changed to the start of the buffer filled with the payload.



```
// Take input message
puts("Enter a message. ");
gets(message);
```



```
0x00010b8c <+0>:     push    {r11, lr}
0x00010b90 <+4>:     add     r11, sp, #4
```



```
0x00010c34 <+168>:   bl      0x10780 <puts@plt>
0x00010c38 <+172>:   sub     r3, r11, #84    ; 0x54
0x00010c3c <+176>:   mov     r0, r3
0x00010c40 <+180>:   bl      0x1072c <gets@plt>
```

How to find the Buffer Address

The starting address of the buffer (containing the payload) can be complex to find. Here are some alternatives:

- **Fuzzing**: make the program running in a similar environment SEGFAULT
- **Iterative Attempts**: testing the exploit exploring the whole address space
- **Unformatted *printf(buffer)***: if present, exploit this other uncommon vulnerability by sending *“%p%p”* as input to the function.

For the sake of the demonstration, *morsecode* will just print the buffer address.

The Attack

With the payload ready, the **attacker** connects to the **victim** service using netcat.

- the payload is injected in the input read buffer, causing it to overflow.
- the control reaches the return address, eventually executing the *shellcode*.

The **attacker** has successfully gained control of a shell and with that of basically the whole board. A simple confirmation could be shutting down the board, but the possibilities are endless (steal data, run malicious programs...)

Summary

- **Buffer Overflow** attacks consist in forcing a program to write outside the intended memory segments, enabling unauthorized code execution.
- Multiple **protection** mechanisms have been developed: DEP, SSP, ASLR...
- However, we have seen that **obsolete systems and bad code** can lead to vulnerabilities that can be easily exploited by attackers.

The demo repository can be found at this link: **GitHub Repo**

Thank you for your time

Group 03