Buffer Overflow Attack:

1. Basic Definition

A **Buffer Overflow** is a software vulnerability that occurs when a program writes more data into a buffer (temporary storage area) than it can hold, causing excess data to overflow into adjacent memory. Attackers exploit this to **crash programs, execute malicious code, or gain unauthorized access.**

Types of Buffer Overflow Attacks:

- Stack-based Buffer Overflow (Most common, exploits the call stack)

- **Heap-based Buffer Overflow** (Exploits dynamically allocated memory)

- Integer Overflow (Caused by incorrect arithmetic operations)

2. How It Works

Step-by-Step Exploitation:

1. Identify a Vulnerable Program

- Targets: C/C++ programs (no built-in bounds checking)

- Common functions: `strcpy()`, `gets()`, `sprintf()`

2. Overflow the Buffer

- Input more data than the buffer can hold (e.g., 500 chars instead of 50).

- Excess data overwrites return addresses, function pointers, or variables.

3. Control Execution Flow

- Overwrite the **return address** to point to attacker-controlled code (shellcode).

- Execute arbitrary commands (e.g., spawn a reverse shell).

Example (Simple C Code Exploit):

```
#include <string.h>
void vulnerable_function(char *input) {
   char buffer[50];
   strcpy(buffer, input); // No bounds checking → Buffer Overflow!
}
int main(int argc, char *argv[]) {
   vulnerable_function(argv[1]);
```

```
    return 0;
}
```

Exploit:

bash

```
./program $(python -c 'print "A" * 100 + "\xef\xbe\xad\xde"')
```

(Overflows buffer and overwrites return address with `0xdeadbeef`)

3. Technical Aspects (For Penetration Testers)

A. Memory Layout (Stack-based Overflow)

- Stack Structure:

| Local Variables | Saved EBP | Return Address | Function Arguments |

- **Overwriting the Return Address:**

  - If input exceeds buffer size, it overwrites the **return address** → Redirects execution.

B. Shellcode Injection

- Shellcode: Small malicious payload (e.g., `/bin/sh` spawner).

- Placement:

  - Injected in the buffer or environment variables.

  - Return address points to shellcode.

C. Exploit Mitigations & Bypasses

Mitigation	How It Works	Bypass Techniques
Stack Canaries	Detects overflow before return	Bruteforce, info leaks
DEP (NX Bit)	Marks stack as non-executable	ROP (Return-Oriented Programming)
ASLR	Randomizes memory addresses	**Bruteforce, info leaks**
Safe Functions	Uses `strncpy()` instead of `strcpy()`	Still exploitable if misused

4. Advanced Techniques (For Pen Testers & Hackers)

A. Return-Oriented Programming (ROP)

- Bypasses DEP/NX by chaining existing code snippets ("gadgets").

- Example:

ROPgadget --binary vuln_program

(Finds gadgets like `pop rdi; ret` for exploit chaining.)

B. Heap Spraying

- Used in browser exploits (JavaScript sprays shellcode in heap memory).

C. Egg Hunting

- Searches for shellcode in memory when space is limited.

D. Fuzzing & Crash Analysis

- Tools:

- AFL (American Fuzzy Lop) – For fuzzing.
- GDB with PEDA – Debugging crashes.
- Immunity Debugger – For exploit development.