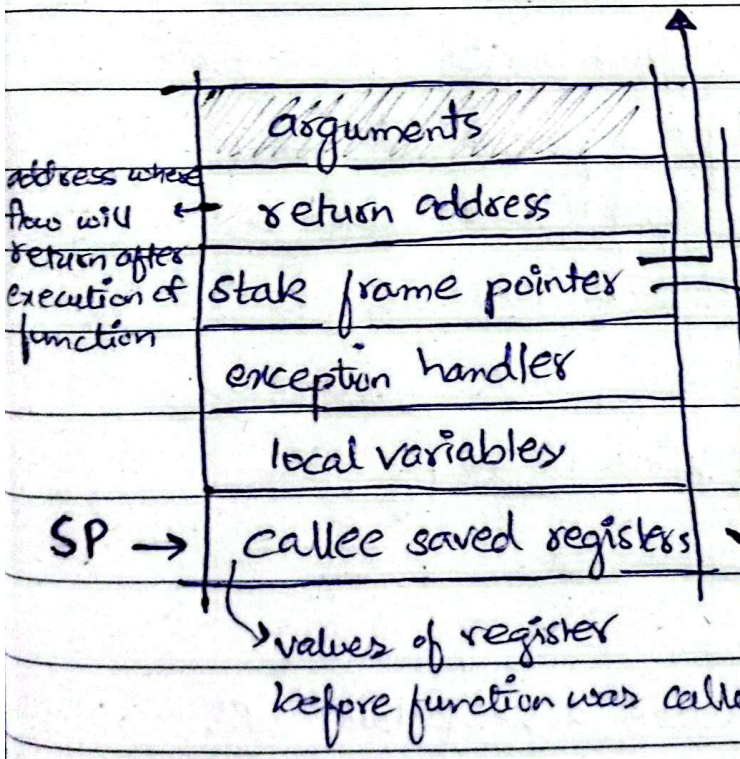
Date _____ Day _____

— CONTROL HIJACKING —

→ Attackers goal is to take over target machine
(eg: web server, email server ..)
↳ Execute arbitary code on target by hijacking application control flow

Examples:

① Buffer Overflow Attacks
② Integer Overflow Attacks
③ Format String vulnerabilities



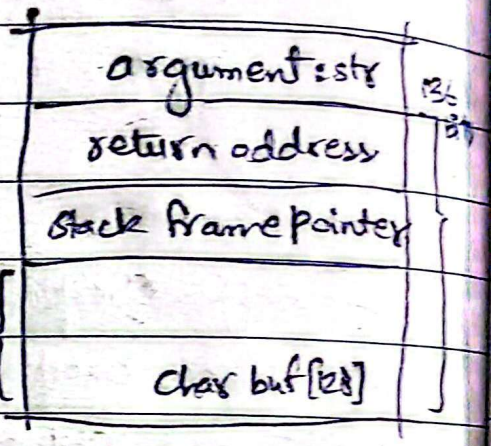
Date _____ Day _____

- **Buffer Overflows**

a func which takes Url from user copy it into buffer and then perform some function

```
void Func (char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```



↳ Now if the *str is 136 bits long then the strcpy() func overflows the "128 bits buffer" & overwrites the values of "stack frame pointer" & "return address"

↳ Now when the function exits it will return to the loc specified in the Url.

→ **Problem:**

↳ No bound Checking in strcpy

→ **Exploit:**

↳ An attacker will create a very long URL which will contain some machine code to execute a shell command, and the value of return addr will point to that machine code. So jese hi function return hoga malicious code will run.

Date _____ Day _____

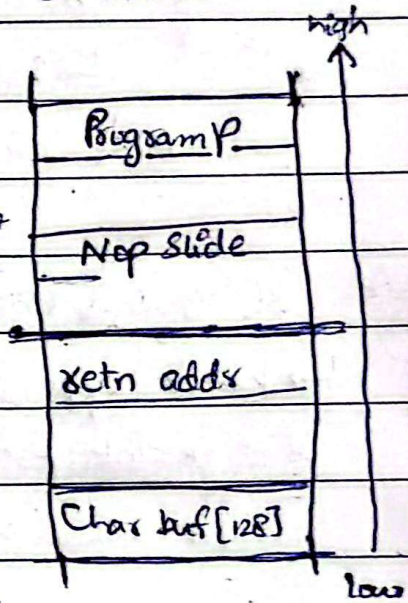
↳ How Return Address is determined?

↳ Attacker guesses approximate stack state when func() is called

↳ Insert many NOPs (No operation command) such as "nop" before Program P

↳ nop execute hota rahega or jese hi Program P par control flow ayega wo execute hojayega

↳ Attack code P runs in stack!



→ The Program P should not starts with '\0' char because since strcpy() were used to copy program P, the null character will cause the copying operation to stop

→ Overflow should not crash program before fun() exists.

→ Unsafe C functions

↳ strcpy (char* dest, cost char* src)

↳ strcat (" , ")

↳ gets (char *s)

↳ scanf (const char *format)

↳ "Safe" libc versions strncpy(), strncat() are also unterminated

Date _____ Day _____

tells processor to do nothing & move on to next instruction

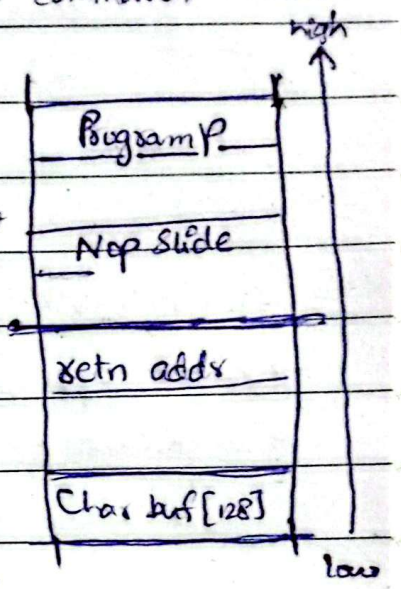
↳ How Return Address is determined?

↳ Attacker guesses approximate stack state when func() is called

↳ Insert many NOPs (No operation command) such as "nop" before Program P

↳ nop execute hota rahega or jese hi Program P par control flow ayega we execute hojayega

↳ Attack code P runs in Stack!



→ The Program P should not starts with '\0' char because since strcpy() was used to copy program P, the null character will cause the copying operation to stop

→ Overflow should not crash program before fun() exists.

→ Unsafe C functions

↳ strcpy (char* dest, const char* src)

↳ strcat (" , ")

↳ gets (char *s)

↳ scanf (const char *format)

↳ "Safe" libc versions strncpy(), strncat() are also not safe → they leave string unterminated

Date _____ Day _____

→ Finding buffer overflows

- Run web server on local machine
- Make malformed requests (ending with "$$$$")
- If webserver crashes
  ↳ search core dump for "$$$$" to find overflow location

• Integer Overflows

↳ What happens when an integer exceeds its maximum value?

- Integers have fixed size
- When the value goes beyond the max
  ↳ it wraps around (modulo arithmetic)

Type	Size	Max Value	Example	Result
char	8 bits	127	0x80 + 0x80 = 256	0
short	16 bits	32767	0xff80 + 0x80 = 65536	0
int	32 bits	$2^{32}-1$	0xffffff80 + 0x80	0

↳ Extra bits are discarded → value wraps to zero

↳ Exploitation

↳ Integer overflow can bypass length checks, leads to buffer overflows, enables memory corruption

Date _____ Day _____

```
void Func (char *buf1, char *buf2, unsigned int len1, len2)
{
    char temp[256];

    if (len1 + len2 > 256) {
        return -1;
    }

    memcpy (temp, buf1, len1)
    memcpy (temp+ len1, buf2, len2);
    do-something(temp)
}
```

if len1 = 0x80 & len2 = 0xffffff80
↳ len1 + len2 = 0 < 256 condition passed

↳ The 2nd memcpy() causes ~~or~~ buffer overflow which can be exploited by an attacker

- **Format String Bug**

↳ A format string vulnerability occurs when user input is used directly as the format string in functions like "printf", "sprintf", "fprintf" etc

Date _____ Day _____

printf (userinput);

↳ programmer expects normal text

↳ Attacker supplies format specifier

→ Format Specifier (%.x, %.n, %.s) make the program:

- Read data from memory
- Write data to memory
- Crash or leak sensitive info

1. Information Disclosure

user_input = "%.x %.x %.x %.x";

↳ prints stack memory

↳ Leaks password, keys, addresses

2. Program Crash

user_input = "%.s %.s %.s";

↳ Tries to read invalid memory

↳ Segmentation fault (DoS)

3. Arbitary Memory Write

user_input = "%.n";

↳ %.n writes no. of printed bytes into memory

↳ Attacker can modify variables or control execution

Date _____ Day _____

it is constant

→ Safe Code ⇒ printf ("%s", usersinput)

→ Exploits

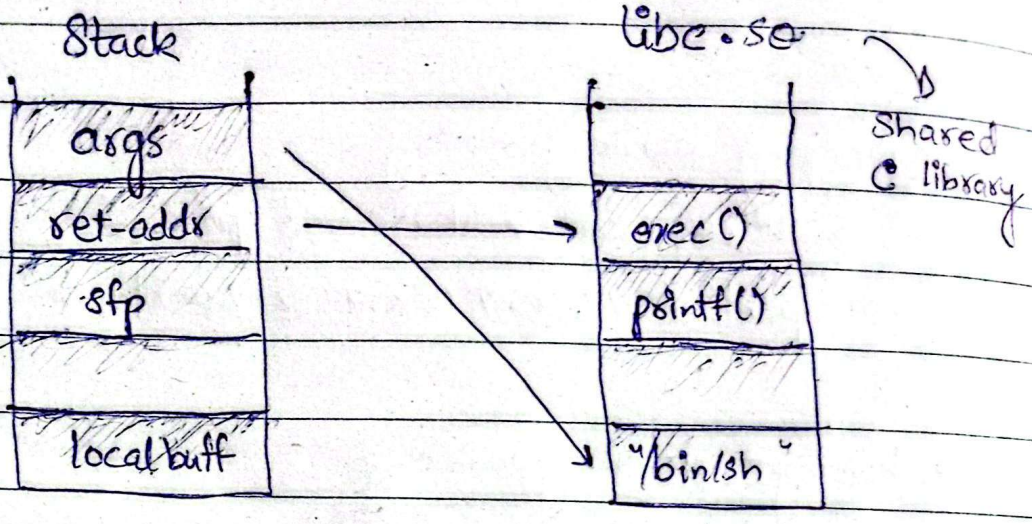
- Dumping arbitary memory
  - ↳ Walk up stack until desired pointer is found
    printf ("%08x.%08x.%08x.%08x|%s|")

- Writing to arbitary memory
  printf ("hello %n", &temp) → write "6" int temp
  print ("%08x.%08x.%08x.%08x %n") → writes 36 into arbitary memory

- Preventing Hijacking Attacks
  - ↳ Audit Software
  - ↳ Rewrite Software in a Type Safe Lang
  - ↳ Add runtime code to detect overflow exploits
    - ↳ Halt process when overflow exploit detected
  - ↳ Concede overflow, but prevent code execution
    - ↳ Making memory as non-executable
      - ↳ Prevent attack code execution by marking stack and heap as non-executable

Date _____ Day _____

- Control Hijacking With On-Memory Code Execution



↳ Instead of running code in stack/heap attacker can point the return address to libc function.

↳ Attacker can also setup multiple stack frames to call chain of libc functions

→ Prevention:

↳ Randomize the location of libc functions

↳ Address Space Layout Randomization (ASLR)

↳ Attacker cannot jump directly to exec()

↳ In windows it is done by default.

↳ Each time sys is rebooted libraries are loaded at diff memory address

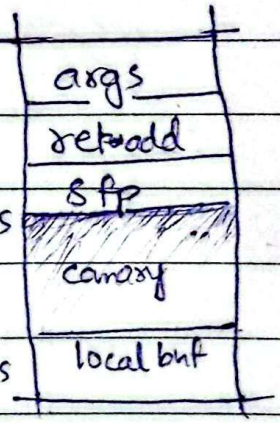
Date _____ Day _____

- **Some Other Attacks : Jit Spraying**
  - ↳ An attack that abuses Just in Time (JIT) compilers to place attacker-controlled code in memory
  - ↳ Force javascript JIT to fill heap with executable shell code.

- **Run-Time Defense**

→ **StackGuard**

- ↳ A runtime protection technique against stack based bufferoverflows
- ↳ It checks stack integrity before a func returns
- ↳ Working:
  - A canary value is placed in each stack frame b/w buffer & ret-add
  - Before func ret, the program verifies the canary
  - If the canary is modified, it means buffer overflow happened, program terminates



- ↳ Types :
  - ① Random Canary
    - A random value is generated at program startup
    - Same random canary is used in all stack frames
    - Size typical 4-8 bytes

Date _____ Day _____

- Checked before func return

(2) Terminator Canary:

- Canary consist of special characters
  ↳ { Null, newline, linefeed, EOF }
- Common string funcs stop copying at these characters
- Prevents overflow using string-based attacks
- ⊖ Doesn't stop non-string memory writes
- ⊖ Less secure than random canary