MONASH University
Information Technology

FIT2093 INTRODUCTION TO CYBERSECURITY

Week 7 Lecture

Software & System Security I

Vulnerabilities & Defenses

S1 2022


Outline

Software & System Security: Vulnerabilities & Defenses

- Basic **Concepts**
- Common **Vulnerabilities** & **Defenses**
  - **buffer overflow: #1 (also related to #3, #5 & #12)**
  - **command injection: #11 (also related to #2, #3, #6, #9)**
  - **integer range overflow: #8**
- Software & System Security **Design Principles**

MONASH University

Basic **Concepts** of **Software Security**


Basic Concepts

Software & System Security: Vulnerabilities & Defenses

- *Recap: security = vs attacks on assets*
- **software** assets: system **data** / **code**
- The Problem: attacks on software
  - via access: read (leakage), (unauthorized changes)
  - via exploiting software functionality (incl change behaviour)
- The Solution: software security
  - **regulate** the access, **manage** the risks/threats of exploitation

What it ain't

- Cyber Security features ≠ Software security
  - e.g. cryptography:
    "*if you think your problem can be solved by cryptography, you don't understand cryptography & you don't understand your problem*"
    [*Bruce Schneier, security guru*]

Q: Give an example of a difference between software security requirements and functional software engineering requirements?

Activity (5 mins)

1) Click the latest link in the Zoom chat
2) Add your question response to the Ed forum
3) Add your “hearts” to your favourite responses

What it ain't

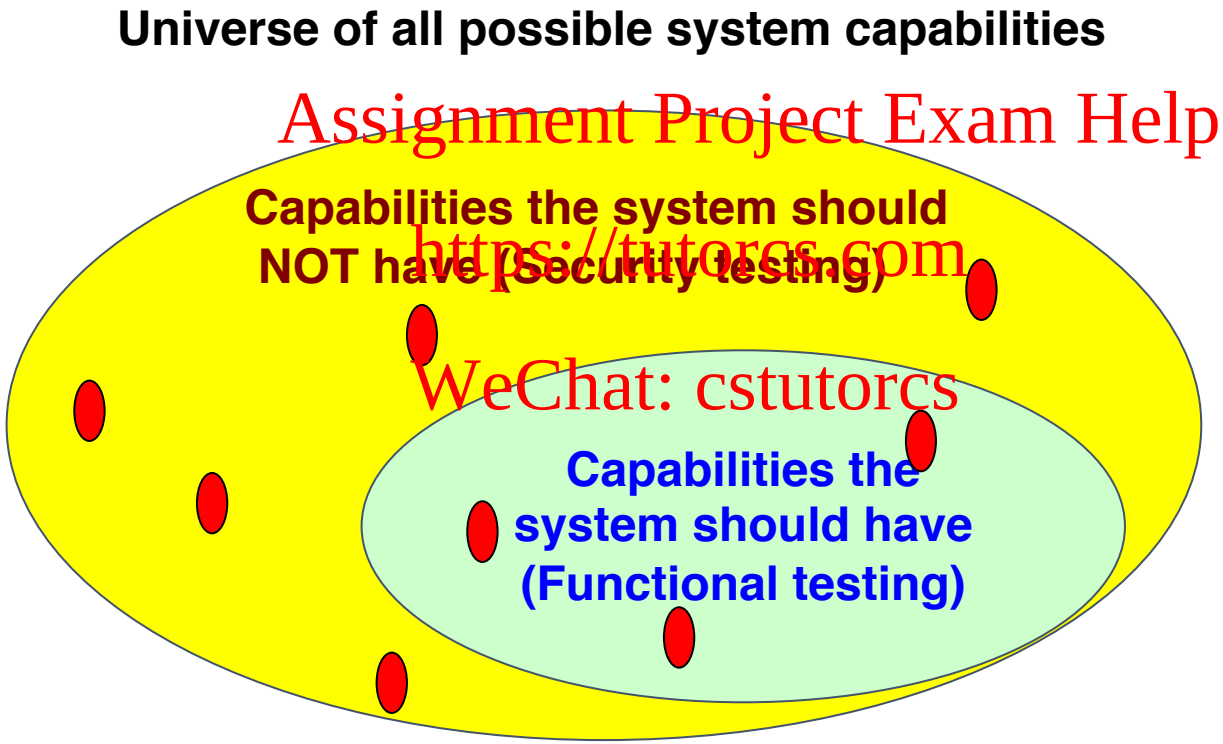
- Software Security ⊆ Software Engineering
- Software engineering lifecycle: requirements, design, code, test, deploy
- conventional SE:
  - quality software = **correct**ness of each stage matching requirements
- real world:
  - quality software = correct, **reliable**, even if attackers try to exploit / attack


Basic Concepts

Software & System Security: Vulnerabilities & Defenses

- Software security
  - quality (correctness) & reliability (robustness)
- **non-functional** security should be **integral** to design, right from start
  - like how testing should be integral

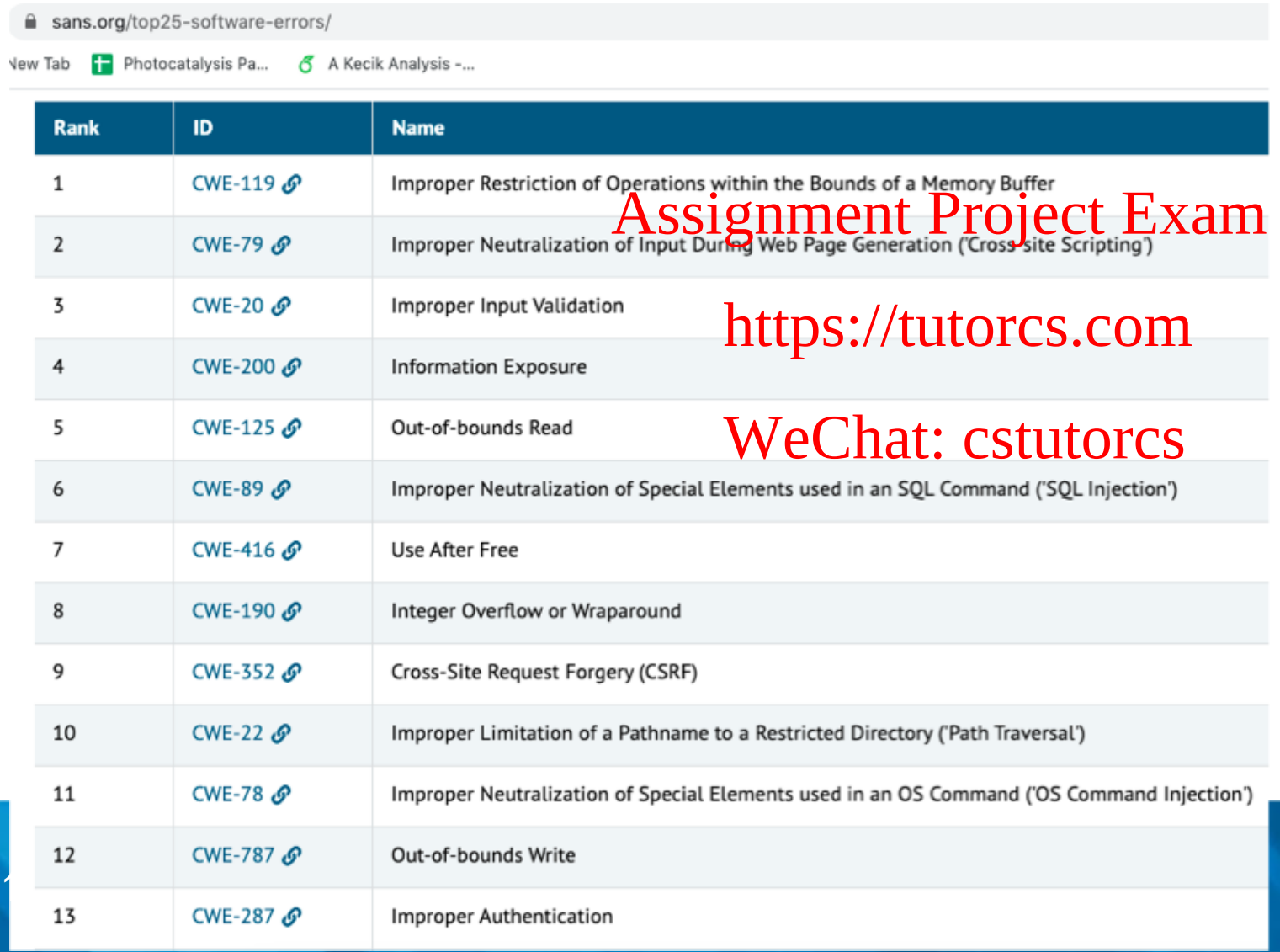
Difference between Functionality Testing and Security Testing



(source: Steven Splain, Testing web security, p. 17)

Top Software Errors

Software & System Security: Vulnerabilities & Defenses



sans.org/top25-software-errors/

New Tab | Photocatalysis Pa... | A Kecik Analysis -...

Rank	ID	Name
1	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-20	Improper Input Validation
4	CWE-200	Information Exposure
5	CWE-125	Out-of-bounds Read
6	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
7	CWE-416	Use After Free
8	CWE-190	Integer Overflow or Wraparound
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
11	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
12	CWE-787	Out-of-bounds Write
13	CWE-287	Improper Authentication

- Common Weakness Enumeration (CWE)/SANS Top 25 Errors [www.sans.org/top25-software-errors/]


Top Software Errors

Software & System Security: Vulnerabilities & Defenses

- 1: Improper Restriction of Operations within the **Bounds** of a Memory **Buffer**
- 2: **Cross-site Scripting** (XSS)
- 3: **Improper Input** Validation
- 4: Information **Exposure**
- 5: **Out-of-bounds Read**
- 6: **SQL Injection**
- 8: **Integer Overflow** or Wraparound
- 10: Improper Limitation of a Pathname to a Restricted Directory ('**Path Traversal**')

- *Q: which vulnerability threatens which security goal: C,I,A + A ?*

Top Software Errors

Software & System Security: Vulnerabilities & Defenses


- 11: Improper Neutralization of Special Elements used in an OS Command ('**OS Command Injection**')
- 12: **Out-of-bounds Write**
- 13: Improper **Authentication**
- 15: Incorrect **Permission** Assignment for **Critical** Resource
- 16: Unrestricted Upload of **File** with **Dangerous** Type
- 18: Improper Control of Generation of Code ('**Code Injection**')
- 20: **Uncontrolled Resource** Consumption
- 24: Improper **Privilege** Management

MONASH University

Buffer Overflow

Buffer Overflow

Software & System Security: Vulnerabilities & Defenses

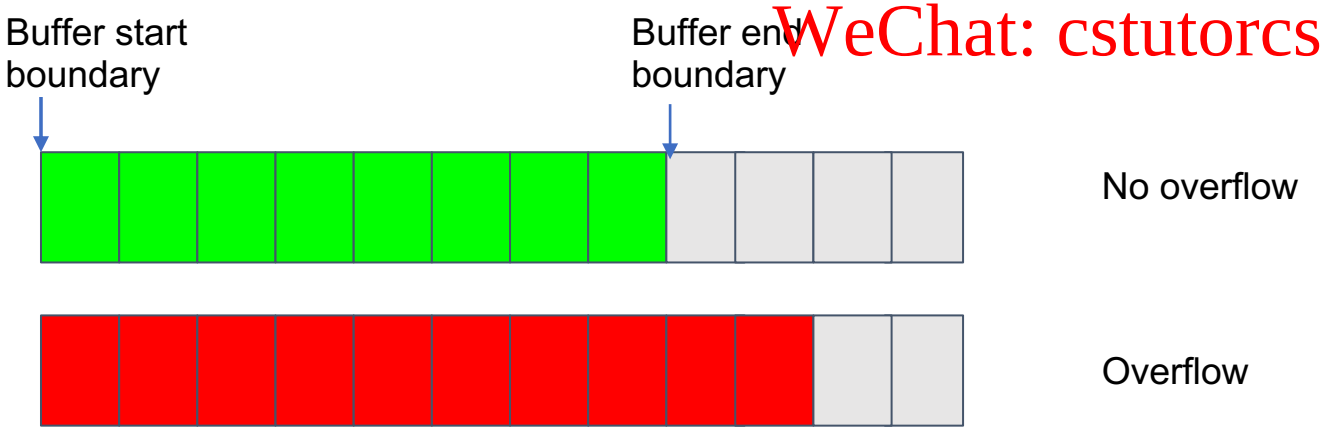
- #1 software error for many years …
- The Vulnerability:
  - Software allocates **buffer** of length N for user **input**
  - then copies input data into buffer without checking input length
- The Exploit:
  - if length(input) > buffer length N:
    - excess input data overwrites memory addresses after buffer

Buffer Overflow: Gist

Software & System Security: Vulnerabilities & Defenses

- *gist*
  - program variable stored in memory, in contiguous manner
  - overwrite beyond boundary, goes into other neighbouring locations

    e.g. buffer length N = 8 but |userInput| = 10



Buffer Overflow

Software & System Security: Vulnerabilities & Defenses

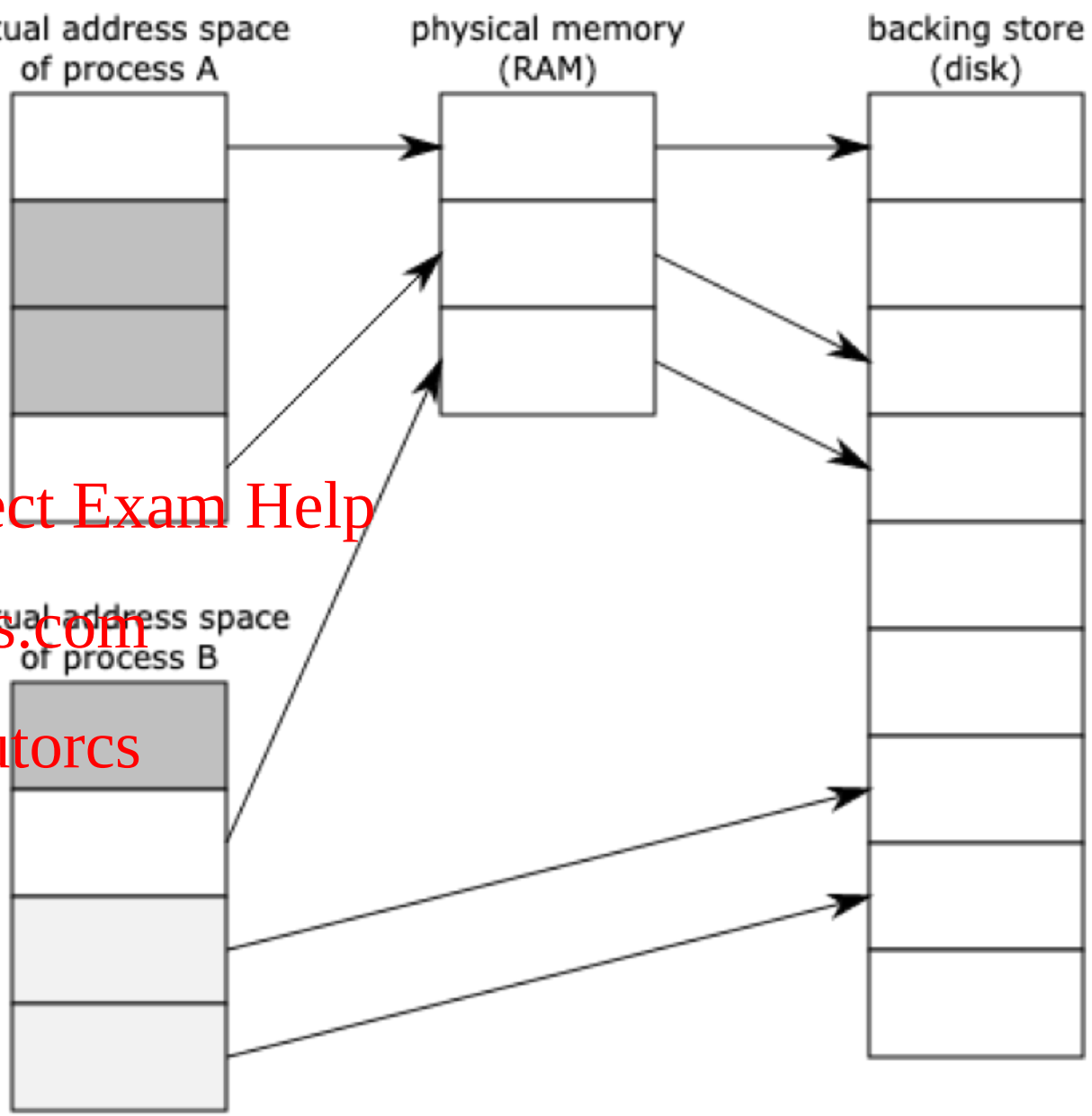
- excess input data overwrites memory addresses after buffer ...
  - normal users: program crash because overwritten with non-relevant data

    vs
  - attackers:
    - Overwrite critical program variables/data [*Example* 1]
    - Overwrite function return address, to point to elsewhere, e.g.
      - desired existing program code [*Example* 2] or
      - code injected by hacker, e.g. a root user shell: “shellcode” [*Lab*]

Processes & Memory

Software Security: Vulnerabilities & Defenses

- for execution, program code&data moved from disk to RAM
- **process** = when a program is executed
- process code&data allocated in RAM/virtual memory, each has an **address**
- memory divided into many segments of different **types**
  - to store code, data, ...
  - our examples relate to stack memory

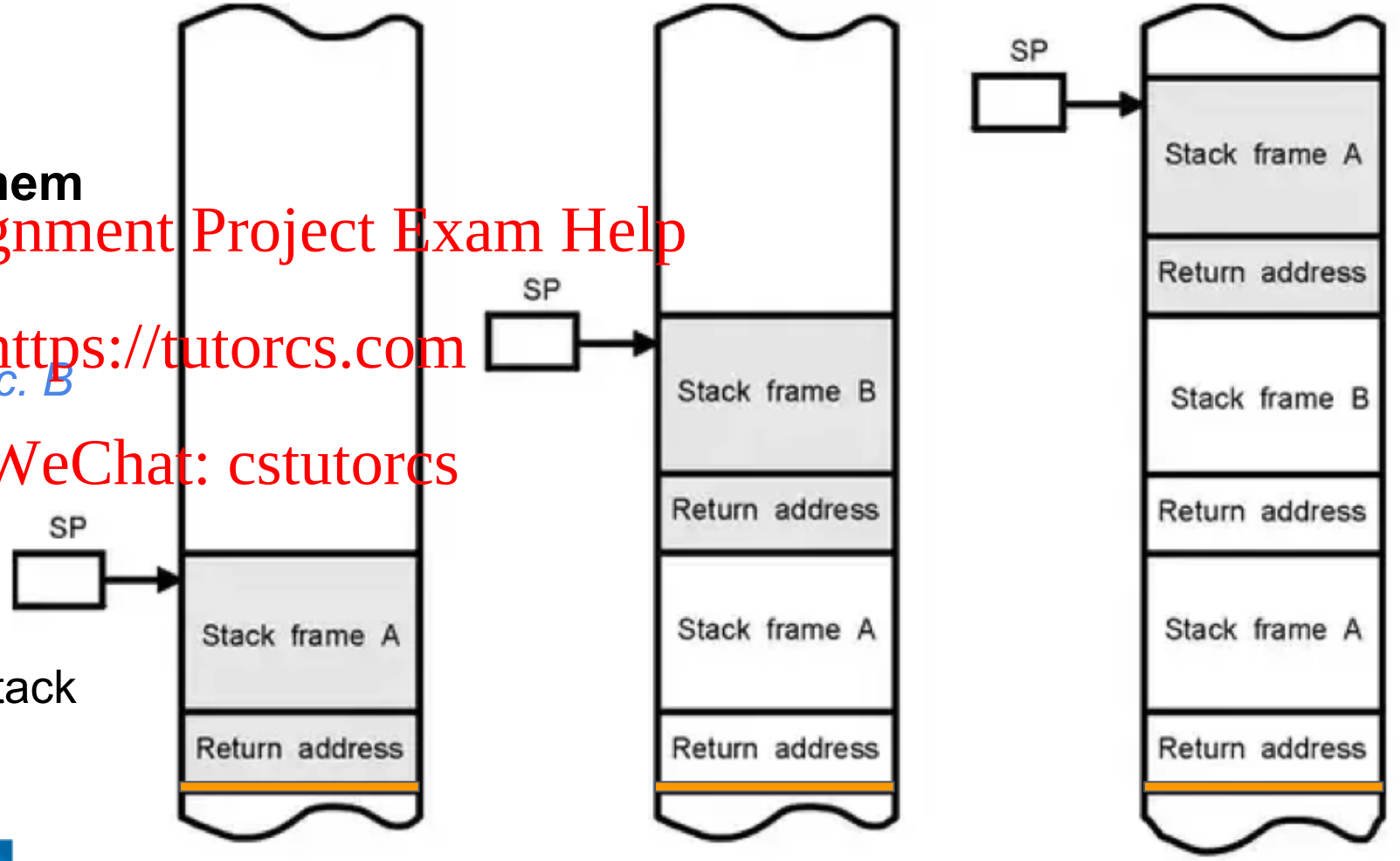


Functions & Stack Memory

Software & System Security: Vulnerabilities & Defenses

- when function called,
  code & data stored in **stack mem**
  e.g. function A called,
  func. B called by func. A,
  func. C called by func. B
- code & data put into stack like putting a stack of papers into a tray: last in first out (LIFO);
  stack pointer (SP) tracks top of stack

  Q: why useful for function calls?



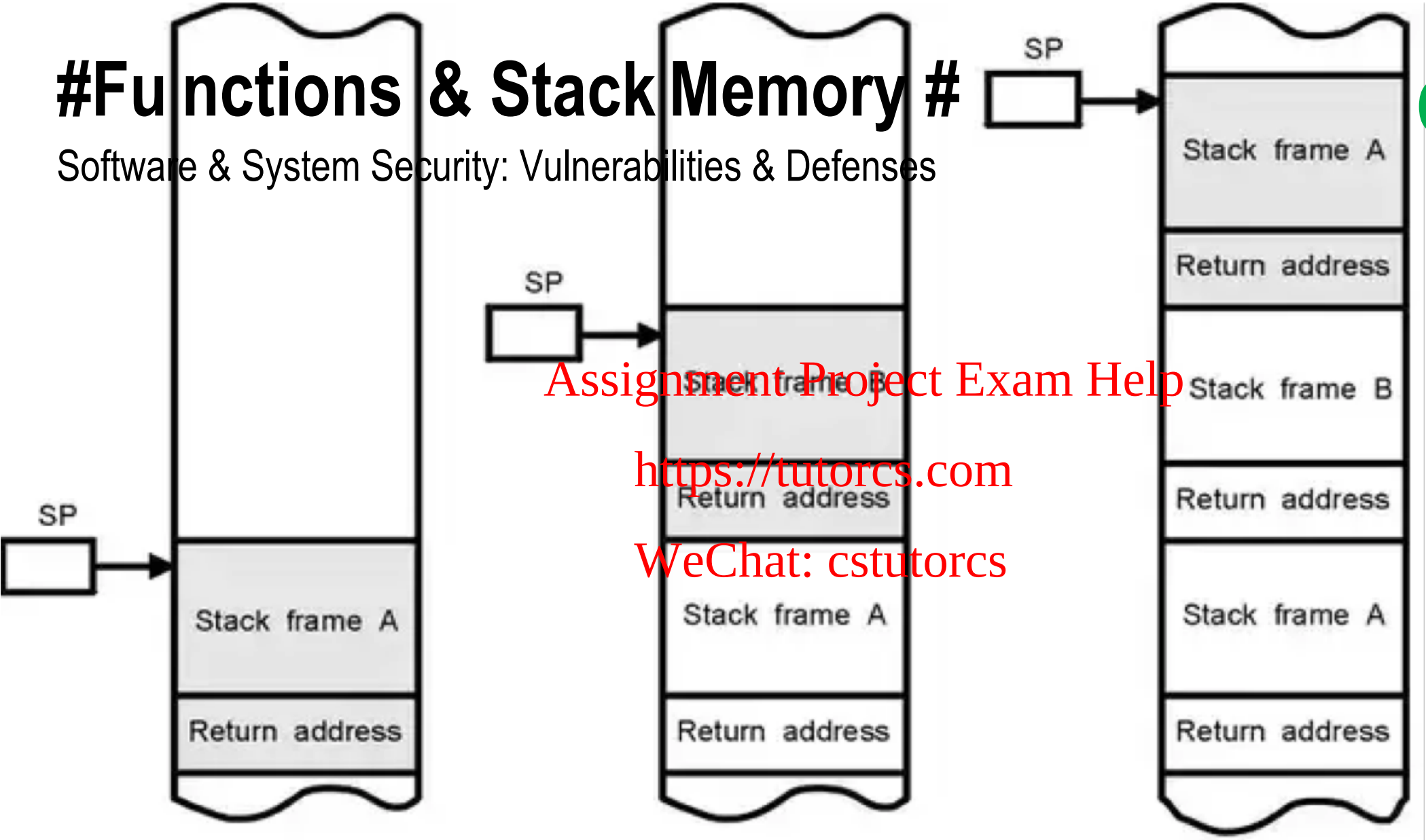
a. The state of the stack during subroutine A

b. The state of the stack during subroutine B

c. The state of the stack during a second call to

#Functions & Stack Memory

Software & System Security: Vulnerabilities & Defenses



a. The state of the stack during subroutine A

b. The state of the stack during subroutine B

c. The state of the stack during a second call to subroutine A

Q: Why is LIFO stack useful for function calls?

Why not FIFO?

Functions & Stack Memory

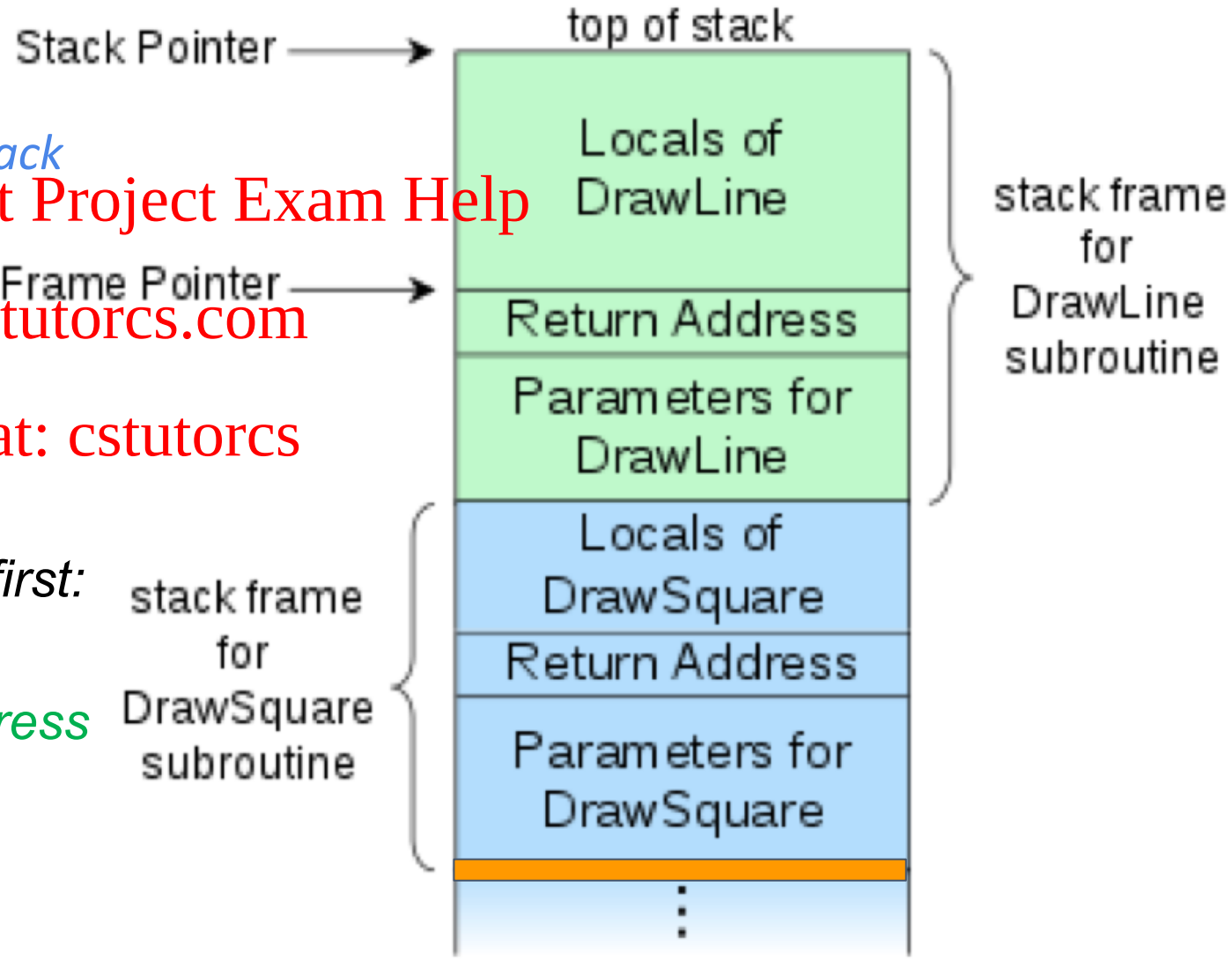
Software & System Security: Vulnerabilities & Defenses

- *e.g.1 DrawSquare calls DrawLine*
  - *DrawSquare code & data saved in stack*
  - *followed by DrawLine*
    - *input parameters*
    - *return address*
    - *local variables*
  - *DrawLine completes execution first*

    so its code/data in stack cleared first:

    LIFO

    Stack grows from high → low address



Buffer Overflow Example 1

Software & System Security: Vulnerabilities & Defenses

- **input** to program: password, via argv[ ] param
- main() function calls check_auth() function
- auth_flag initialized to 0 (wrong pwd by default)
- buffer: password_buffer of 16 chars
- input password **copied** to buffer with strcopy, without length check

```
int check_auth(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    strcpy(password_buffer, password);
    if(strcmp(password_buffer,"mypasswd")==0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
  if(check_auth(argv[1]))
    {
printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
printf("      Access Granted.\n");
printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    }
    else
        printf("\nAccess Denied.\n");
}
```

Buffer Overflow Example 1

Software & System Security: Vulnerabilities & Defenses

- input password **copied** to buffer with strcopy, without length check

Q: How could a malicious user of this program cause a buffer overflow? How could it help get access granted?

```
int check_auth(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    strcpy(password_buffer, password);
    if(strcmp(password_buffer,"mypasswd")==0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
   if(check_auth(argv[1]))
   {
printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
printf("      Access Granted.\n");
printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   }
   else
        printf("\nAccess Denied.\n");
}
```

Buffer Overflow Example 1

Software & System Security: Vulnerabilities & Defenses

- strcmp **compares** two strings, returns **0 if equal**
  - 0 = false, non-zero = true
  - 0 means buffer value = correct passwd
    - auth_flag set to 1
- auth_flag returned to main func
- if(auth_flag):
  - 0 = false, non-zero = true


```
int check_auth(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    strcpy(password_buffer, password);
    if(strcmp(password_buffer,"mypasswd")==0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
  if(check_auth(argv[1]))
    {
printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-
\n");
    }
   else
        printf("\nAccess Denied.\n");
}
```

Buffer Overflow Example 1

Software & System Security: Vulnerabilities & Defenses

Buffer Overflow example 1 [HTAE]:

Overwrite critical program variables

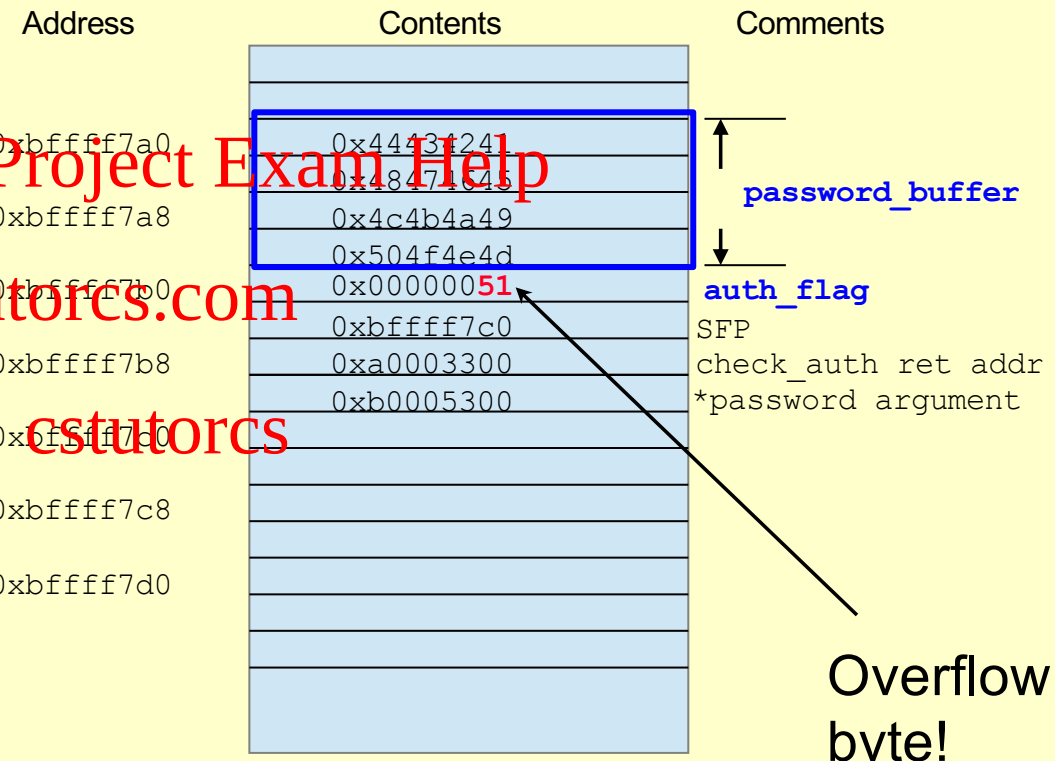
auth_overflow.c:

```
int check_auth(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    strcpy(password_buffer, password);
    if(strcmp(password_buffer,"mypasswd")==0)
       auth_flag = 1;
    return auth_flag;
}


int main(int argc, char *argv[]) {
   if(check_auth(argv[1]))
   {
         printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
         printf("      Access Granted.\n");
         printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   }
   else
         printf("\nAccess Denied.\n");

}
```


Memory Contents after strcpy of password "ABCDEFGHIJKLMNOPQ" = 41H,42H, … 4FH, 50H, 51H



Buffer allocated for passwords up to 16 chars length:
longer passwords overflow into auth_flag local variable

Buffer Overflow Example 1

Software & System Security: Vulnerabilities & Defenses

- *assumption*:
  - auth_flag only becomes non-0 if password correct, as a result of the strcmp
  - so if non-0,
    - must be passwd correct
- but can exploit buffer overflow
  - to make auth_flag be non-0
- **input**: ABCDEFGHIJKLMNOPQ
  - give password longer than 16
  - Q is written into next space, i.e. auth_flag

```
int check_auth(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    strcpy(password_buffer, password);
    if(strcmp(password_buffer,"mypasswd")==0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
  if(check_auth(argv[1]))
    {
printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-
\n");
    }
   else
        printf("\nAccess Denied.\n");
}
```

Buffer Overflow Example 2

Software & System Security: Vulnerabilities & Defenses

What if password_buffer is stored **after** local vars?
buffer overflow cannot overwrite auth_flag variable
Solution: attacker overwrites **return address** to jump to desired code

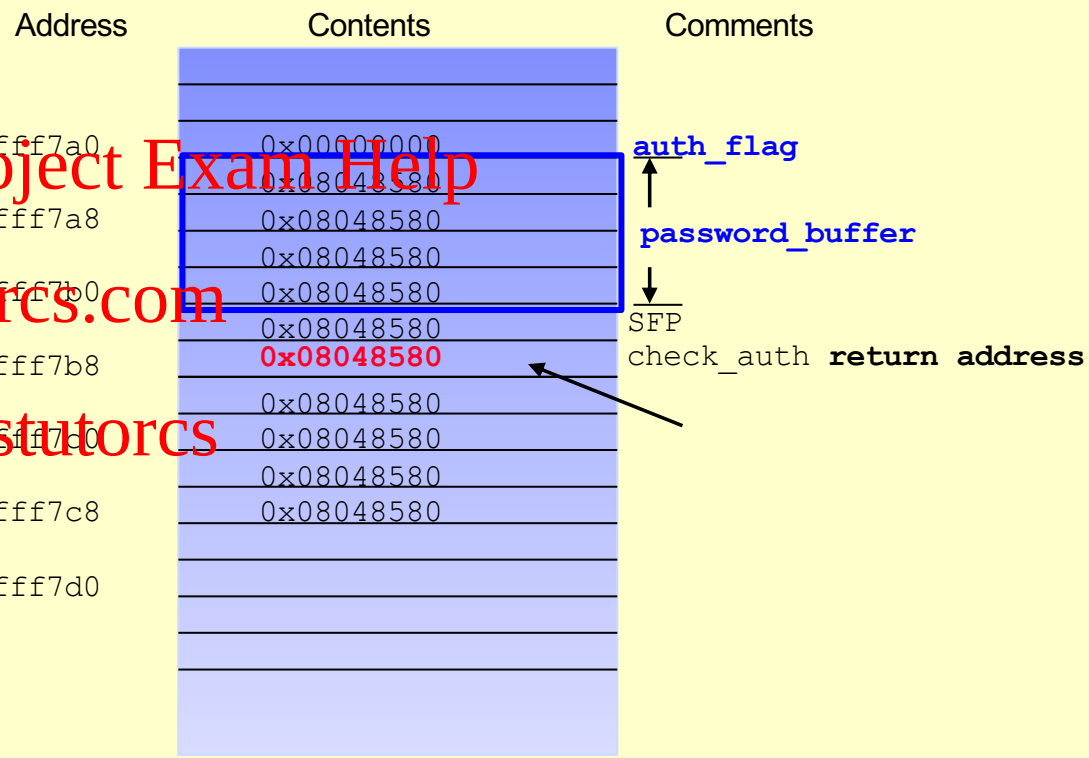
auth_overflow2.c:

```
int check_auth(char *password) {
   char password_buffer[16];
   int  auth_flag[1];
   auth_flag[0] = 0;
   strcpy(password_buffer, password);
   if(strcmp(password_buffer, "mypasswd") ==
0)
        auth_flag[0] = 1;
   return auth_flag[0];
}

int main(int argc, char *argv[]){
  if(check_auth(argv[1]))
  {...
```

attacker wants to return exec. here: address 0x8048580

Memory Contents after strcpy of password
"\x80\x85\x04\x08" x 10



Attacker uses repeated return address to avoid needing to know exact location of check_auth return address relative to buffer start

Buffer Overflow Example 2

Software & System Security: Vulnerabilities & Defenses

- **disassemble** the program using a **debugger**: get assembly language

```
0x0804853d <+1>:     mov    ebp,esp
0x0804853f <+3>:     and    esp,0xfffffff0
0x08048542 <+6>:     sub    esp,0x10
0x08048545 <+9>:     cmp    DWORD PTR [ebp+0x8],0x1
0x08048549 <+13>:    jg     0x804856c <main+48>
0x0804854b <+15>:    mov    eax,DWORD PTR [ebp+0xc]
0x0804854e <+18>:    mov    eax,DWORD PTR [eax]
0x08048550 <+20>:    mov    DWORD PTR [esp+0x4],eax
0x08048554 <+24>:    mov    DWORD PTR [esp],0x8048661
0x0804855b <+31>:    call   0x8048380 <printf@plt>
0x08048560 <+36>:    mov    DWORD PTR [esp],0x0
0x08048567 <+43>:    call   0x80483c0 <exit@plt>
0x0804856c <+48>:    mov    eax,DWORD PTR [ebp+0xc]
0x0804856f <+51>:    add    eax,0x4
0x08048572 <+54>:    mov    eax,DWORD PTR [eax]
0x08048574 <+56>:    mov    DWORD PTR [esp],eax
0x08048577 <+59>:    call   0x80484dc <check_authentication>
0x0804857c <+64>:    test   eax,eax
0x0804857e <+66>:    je     0x80485a6 <main+106>
0x08048580 <+68>:    mov    DWORD PTR [esp],0x8048677
0x08048587 <+75>:    call   0x80483a0 <puts@plt>
```


check if return value = 0
if = 0, go elsewhere
else continue

Buffer Overflow Example 2

Software & System Security: Vulnerabilities & Defenses

- **disassemble** the program using a **debugger**: get assembly language

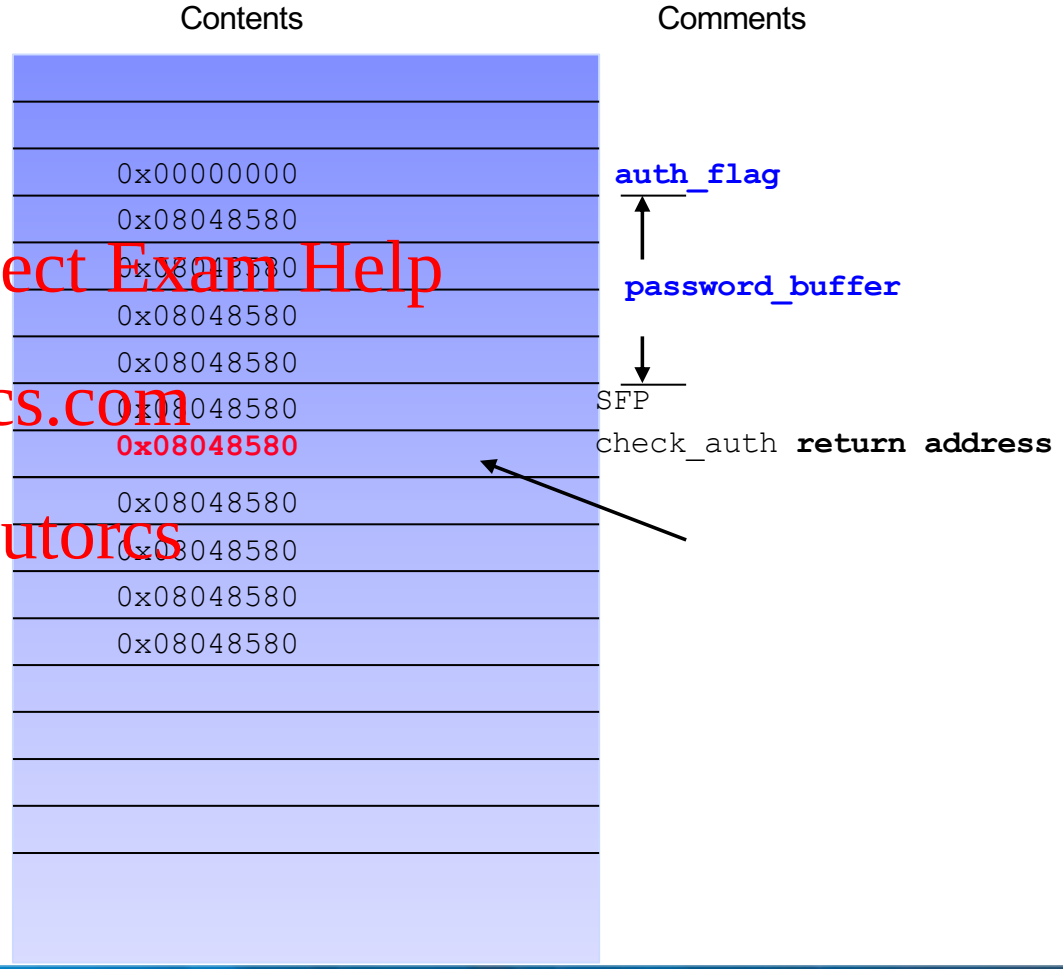
```
0x0804853d <+1>:     mov    ebp,esp
0x0804853f <+3>:     and    esp,0xfffffff0
0x08048542 <+6>:     sub    esp,0x10
0x08048545 <+9>:     cmp    DWORD PTR [ebp+0x8],0x1
0x08048549 <+13>:    jg     0x804856c <main+48>
0x0804854b <+15>:    mov    eax,DWORD PTR [ebp+0xc]
0x0804854e <+18>:    mov    eax,DWORD PTR [eax]
0x08048550 <+20>:    mov    DWORD PTR [esp+0x4],eax
0x08048554 <+24>:    mov    DWORD PTR [esp],0x8048661
0x0804855b <+31>:    call   0x8048380 <printf@plt>
0x08048560 <+36>:    mov    DWORD PTR [esp],0x0
0x08048567 <+43>:    call   0x80483c0 <exit@plt>
0x0804856c <+48>:    mov    eax,DWORD PTR [ebp+0xc]
0x0804856f <+51>:    add    eax,0x4
0x08048572 <+54>:    mov    eax,DWORD PTR [eax]
0x08048574 <+56>:    mov    DWORD PTR [esp],eax
0x08048577 <+59>:    call   0x80484dc <check_authentication>
0x0804857c <+64>:    test   eax,eax
0x0804857e <+66>:    je     0x80485a6 <main+106>
0x08048580 <+68>:    mov    DWORD PTR [esp],0x8048677
0x08048587 <+75>:    call   0x80483a0 <puts@plt>
```


if = 0, go elsewhere
else continue

Buffer Overflow Example 2

Software & System Security: Vulnerabilities & Defenses

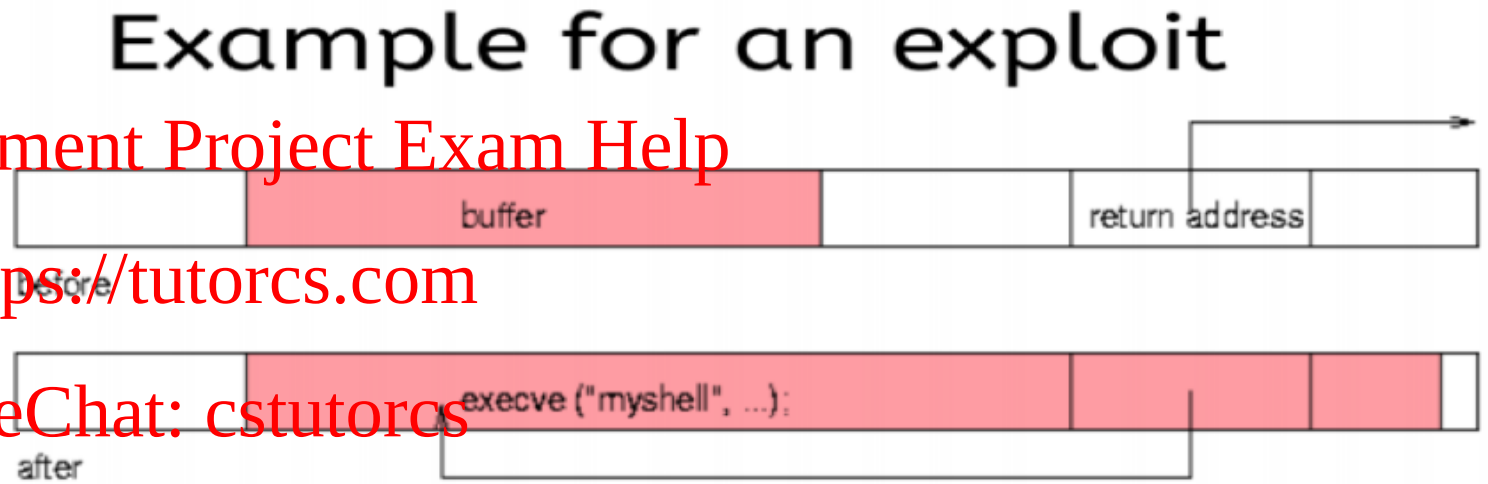
- exploit buffer overflow
  - to **change** return address to where program goes if passwd correct
- overflow 0x08048580 x(*manytimes*)
  - password_buffer will overflow into other locations incl return address
  - when function returns, will go to that location i.e. same as where program goes if passwd correct



Buffer Overflow Example 2

Software & System Security: Vulnerabilities & Defenses

- before
  - red: buffer
- after
  - red: input written into buffer & nearby region incl return address



Buffer Overflow: Defenses

Software & System Security: Vulnerabilities & Defenses

- Problem: user input not checked
- Solution:
  - check that bounds of allocated memory are not exceeded
- Lessons:
  - Don't trust user input
  - Defense principle: **validate all inputs**
  - Don't use unsafe function calls
    - e.g. gets()/strcpy() in C **do not** check user input size

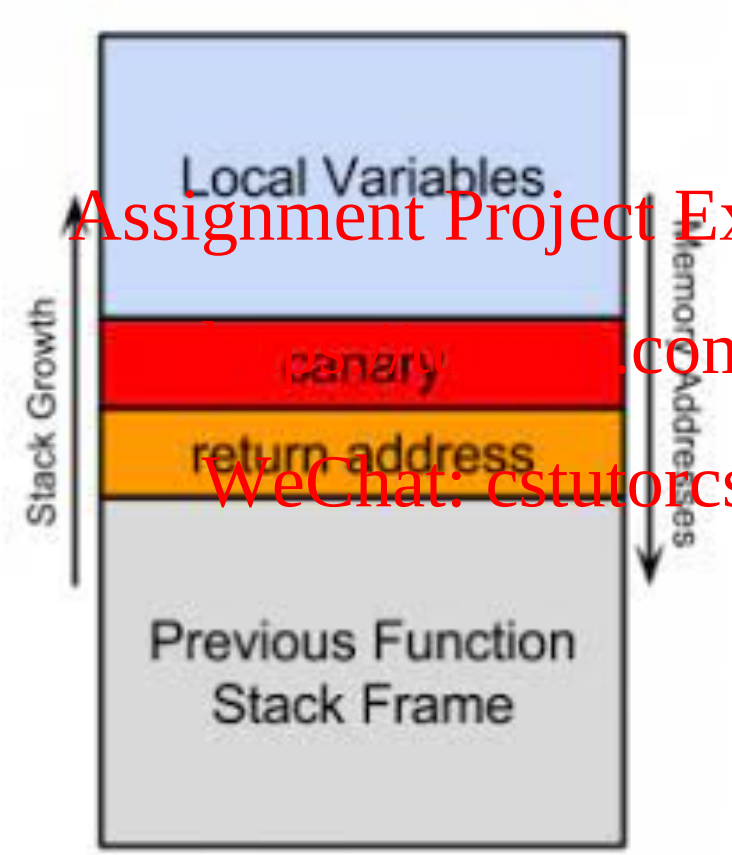
Buffer Overflow: Defenses

Software & System Security: Vulnerabilities & Defenses

- **Defenses** by operating systems:
  - e.g. address randomization
- **Defenses** by compilers:
  - e.g. use canaries:
    - known values inserted into stack between buffer & other critical data
    - 1st to be corrupted if buffer overflow (since right after it)
    - Run time tests for stack integrity
      - verify their integrity prior to function return
    - e.g. StackGuard (compiler extension) & ProPolice for gcc compiler

Buffer Overflow: Defenses

Software & System Security: Vulnerabilities & Defenses



Command Injection

Command Injection

Software & System Security: Vulnerabilities & Defenses

- *Gist*
  - **User input** includes extra **command**
  - **Code does not check user input to prevent/remove** extra commands

- common types
  - C Format String vulnerabilities
  - OS command injection
  - *SQL Injection* [*later weeks*]
  - *Cross-Site Scripting (XSS)* [*later weeks*]

Command Injection Example

Software & System Security: Vulnerabilities & Defenses

- system code put into variable command
- system( ) runs a command in the host env

```
int main ()
{
    char command[50];

    strcpy(command, "ls -l");
    system(command);

    return(0);
}
```

- passed to & run in host environment

Q: Suppose we change the red line in the code to:

```
strcpy(command, "ls -l; cat file.txt");
```

What would happen if we run this program?

Command Injection Example

Software & System Security: Vulnerabilities & Defenses

- Normal usage: `check ls`
  - results in system command:
    - `time ls`
    - (*reports execution time of `ls`*)
- Malicious exploit: `check ls; cat /etc/shadow`
  - results in system command:
    - `time ls; cat /etc/shadow`
    - (*reports execution time of `ls`* and then displays *contents of `/etc/shadow` password file!*)
  - appended extra commands via user input!
- [https://www.owasp.org/index.php/Command_Injection]

`Check.c` program code:

```
int main(int argc, char **argv)
{
    char command[256];
    ...
    strcat(command, "time ");
    strcat(command, argv[1]);
    system(command);
    return 0;
}
```

e.g. Running 'check ls' will cause argv[0] = 'check', argv[1] = 'ls', which means 'time ls' will be run at command prompt

Command Injection Example

Software & System Security: Vulnerabilities & Defenses

- The Problem:
  - **no input validation**, can append any malicious command via semicolon ;
  - original application still runs, but extra commands run after it

- gist: program expects to get only data input and data appended to program to be run,
- but extra commands appended within the data field, and all executed one after another via semicolon ;

- extra command will be run with privileges of program owner (e.g. root)

```
int main(int argc, char **argv)
{
    char command[256];
    ...
    strcat(command, "time ");
    strcat(command, argv[1]);
    system(command);
    return 0;
}
```

Command Injection: Defenses

Software & System Security: Vulnerabilities & Defenses

- sanitize all user input data
- avoid passing user-given arguments to OS programs
- use a command execution API that takes the command & arguments as separate parameters:
  - Command parameters not controlled by user input but by app code
  - Argument parameter based on user input: cannot be interpreted by system as a command!
- Remove potentially damaging characters
- Whitelist: be **explicit** on what's permitted, don't assume

MONASH University

Integer Range Overflow

Integer Range Overflow

Software & System Security: Vulnerabilities & Defenses

- **Integer variables: commonly for array indexing & arithmetic**
  - e.g. typically on a 32-bit machine in C:

- **`unsigned int` : 32 bit**
  - possible values: [0, 2^{32}–1] (hex [`0x00000000,0xffffffff`])
- **`unsigned short` : 16 bit**
  - possible values: [0, 2^{16}–1] (hex [`0x0000,0xffff`])

- **`int` : 32 bit**
  - range: [–2^{31}, 2^{31}–1] (hex [`0x10000000,0x7fffffff`])
  - MSBit = sign bit: Negative numbers (–2^{31} <= `x < 0`) represented as the number `x (mod` 2^{32}`)`

Q: Suppose we have $x = 2^{16} - 1 = 65535$ in C code for an `unsigned short` variable x, and we run the increment code

`x = x + 1;`

What will be the value of x now?

Integer Range Overflow

Software & System Security: Vulnerabilities & Defenses

- **`int` : 32 bit**
  - range: $[-2^{31}, 2^{31}-1]$ (hex [`0x10000000,0x7fffffff`])
  - MSBit = sign bit: Negative numbers ($-2^{31} <= x < 0$) represented as the number $x \pmod{2^{32}}$
  - e.g. –1 represented as $2^{32}-1$ = `0xffffffff`
  - e.g. for 3 bit numbers:
    - +ve: 000, 001, 010, 011 are 0, 1, 2, 3
    - –1 represented as $2^3-1 = 8-1 = 7 = 111$
    - –2 represented as $2^3-2 = 8-2 = 6 = 110$ …
    - so -4, –3, –2, –1, 0, 1, 2, 3
      = $2^3+(-4)$, $2^3+(-3)$, $2^3+(-2)$, $2^3+(-1)$, 2^3+0, 2^3+1, 2^3+2, 2^3+3

Integer Range Overflow

Software & System Security: Vulnerabilities & Defenses

- –3, –2, –1, 0, 1, 2, 3
  = 2^3+(–3), 2^3+(–2), 2^3+(–1), 2^3+0, 2^3+1, 2^3+2, 2^3+3
- Integer variables can overflow if one writes a value larger or smaller than the range of possible values for the integer type
- Maliciously-chosen integer overflows can also change program behaviour, and can be exploitable by attackers

Integer Range Overflow Example

Software & System Security: Vulnerabilities & Defenses

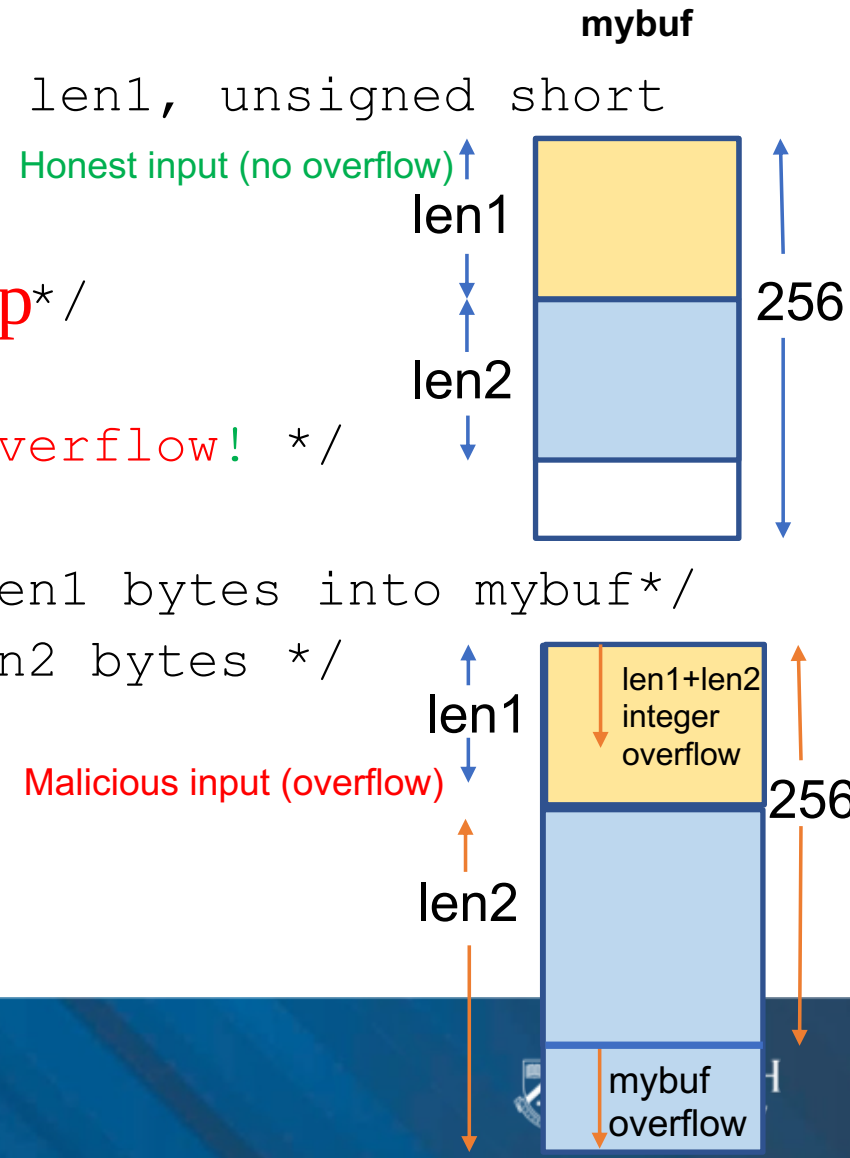
- When an arithmetic expression involving an integer overflows the `max` for that type, **the result** (*with most compilers*) **is reduced modulo (`max+1`)**
- e.g
  - `unsigned short num = 0xffff; // 0xffff = 65535 = max`
  - `num = num + 2; (addition overflow)`
  - `printf("num = 0x%x\n", num);`
  - Output: `num = 0x01 // (65535+2) mod (max+1) = 65537 mod 65536 = 1`

  - `unsigned short num_mul = 0x4000; // 0x4000 = 65536/4`
  - `num = num * 4; (multiplication overflow)`
  - `printf("num = 0x%x\n", num);`
  - Output: `num = 0x0 // 65536 mod 65536 = 0`

Integer Range Overflow Example

Software & System Security: Vulnerabilities & Defenses

```
int catvars(char *buf1, char *buf2, unsigned short len1, unsigned short
len2)
{
char mybuf[256]; /* allocate 256 byte buffer mybuf */
if((len1 + len2) > 256) /* overflow check flawed
                           because of len1+len2 sum overflow! */
      { return -1; }
memcpy(mybuf, buf1, len1);      /*copy len1 bytes into mybuf*/
memcpy(mybuf+len1, buf2, len2);     /*then copy len2 bytes */
doSomeStuff(mybuf);
return 0;
}
```

Integer Range Overflow Example

Software & System Security: Vulnerabilities & Defenses

- How attacker bypasses buffer overflow check using integer overflow:
  - Call `catvars()` function with attacker supplied length inputs:
    - `len1 = 0x000a` = **10**
    - `len2 = 0xfffc` (= $2^{16} - 4$) = **65,532**
  - Length sum overflow:
    - `len1 + len2` should be (if no overflow) = $2^{16} + 6$
    - Due to `unsigned int` overflow, it becomes `(len1 + len2) mod 2^16 = 6`
    - Passes the buffer overflow test i.e. (`if((len1 + len2) > 256)`)
      - because `(len1 + len2)` overflows to `(len1 + len2) mod 2^16 = 6 < 256`
      - Yet `len2` > 256 byte mybuf length
      - → Second memcpy() call to copy len2 bytes overflows mybuf!

Integer Range Overflow: Defenses

Software & System Security: Vulnerabilities & Defenses

- validate inputs:
  - ensure **no arithmetic overflow**


```
int catvars(char *buf1, char *buf2, unsigned int
len1, unsigned int len2)
{
char mybuf[256];

if ( (len1 > 256) || (len2 > 256) )
{ return -1; /* error: integer overflow attempted!
                */}
/* now we know that len1 + len2 < 512 < 65536, so
   len1+len2 won't overflow */

if((len1 + len2) > 256) /* buffer overflow check */
   { return -1; /* error: buffer overflow attempted
   */ }

...
```

MONASH University

Software Security
Design Principles

Design Principle: Least Privilege

Software & System Security: Vulnerabilities & Defenses

- any given privilege should be granted:
  - to the **least** amount of code segment and
  - for the **least** amount of time necessary
- **unless** absolutely **essential**, else **don't**:
  - (let users) run programs with elevated privileges
  - grant **administrative access** to user programs

- Separation of privilege: erect **as many walls around** code parts so failure of one part doesn't create issues on other parts of the system

Design Principle: Input Validation

Software & System Security: Vulnerabilities & Defenses

- Note: Most security vulnerabilities are due to malformed data by user to program
  - e.g. Buffer overflow, Command injection, SQL injection, XSS, etc.
- Look for all data and accept only if all are valid
  - Never trust user input
- Data must be validated as it crosses the boundary between untrusted & trusted environments
  - **Trusted** data: data that you or an entity you explicitly trust has complete control over;
  - **Untrusted** data refers to everything else
- Typically incorporate code that does "**white list**" and/or "**black list**" **validation**
  - **White list**: list of valid inputs e.g. structured data fields (date etc.)
  - **Black list**: inputs that are not accepted in an attempt to detect attack characters and patterns e.g. metacharacters

White List vs Black List

- **White list**: list of valid inputs e.g. structured data fields (date etc.)
- **Black list**: inputs that are not accepted in an attempt to detect attack characters and patterns e.g.

Q: In general, which method of input validation should be preferred for security, white list or black list, and why?

Design Principle: Fail Securely

Software & System Security: Vulnerabilities & Defenses

- Software will fail, cannot avoid, plan for it
  - Fail to a secure mode (e.g. exception handling: secure defaults)
  - be able to **detect** failures (eg, logs)
  - be able to **recover** from failures (eg, data backup)

- do not disclose detailed error info to the user (or hacker)
  - display **just enough** info to notify user that an error occurred & what to do about it
  - capture the detailed error info in a log file

Insecure Exemption Handling Example

Software & System Security: Vulnerabilities & Defenses

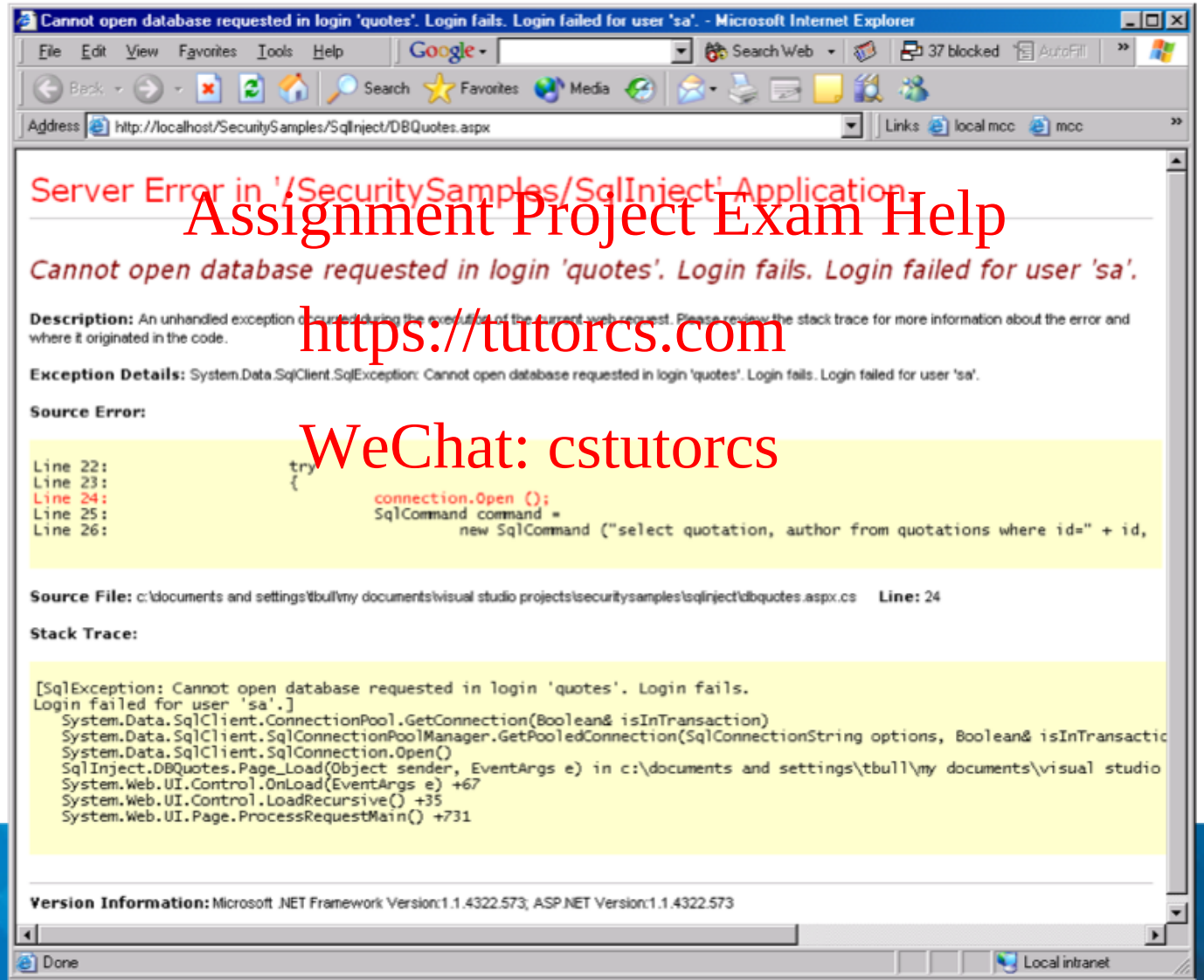
```
// INCORRECT exception handling
DWORD dwRet = OpenFile(…);
Accessgranted = true // default is access
granted
if (dwRet == ERROR_1) {
        Accessgranted = false
        // deny access if specific error
occurs
} else {
        // Default access granted.
        // Perform task.
}
```

```
// CORRECT exception handling
DWORD dwRet = OpenFile(…);
Accessgranted = false // default is
access denied
if (dwRet == NO_ERROR) {
        // All OK.
        Accessgranted = true
        // Perform task.
} else {
        // Default access denied
        // Inform user access denied.
}
```


- **Secure default**: Safer to check for successful code to grant access (default is deny) rather than check for specific error to deny access
- program may return other error codes not anticipated!

Insecure Failure Disclosure Example

Software & System Security: Vulnerabilities & Defenses



Design Principle: Reduce Attack Surfaces

- Minimize Attack Surface: all the different points where attacker could get into (infiltrate) into / get data out (exfiltrate) of system
  - e.g. user input fields, protocols, interfaces, services
- Do not install all features & capabilities by default
  - if a feature is installed by default, make sure that it operates under the principle of least privilege

Other Design Principles

- Defense in depth: avoid single points of failure
- Keep design small & simple
- Open design i.e. don't depend on security via obscurity
- Backward compatibility is dangerous
- Assume external systems/entities are insecure
- Authorise after authentication
- Only receive control instructions from trusted sources
- …

Further Reading

- Chapter 10 & 11 of the textbook: Computer Security: Principles and Practice" by William Stallings & Lawrie Brown, Prentice Hall, 2015
- [HTAE] J. Erickson, "Hacking: The Art of Exploitation", 2008. (online access via Monash Library).
- G McGraw, Software Security, Addison-Wesley Software Security Series, 2006. Copy available at the Monash library.
- https://cwe.mitre.org/ - Common Weakness Enumeration (A community-developed list of publicly known types of software and hardware vulnerabilities)
- https://cve.mitre.org/ - A database of publicly known cybersecurity vulnerabilities in specific products