Lecture 9: Software Security (Part III)

9.1 Time-of-Check Time-of-Use (TOCTOU) race condition

9.2 Defense and Preventive Measures:

- 9.2.1 Filtering (input validation)
- 9.2.2 Using safer functions
- 9.2.3 Bounds checking and type safety
- 9.2.4 Memory protection (randomization, canaries)
- 9.2.5 Code inspection (taint analysis)
- 9.2.6 Testing
- 9.2.7 The principle of least privilege
- 9.2.8 Patching (keeping up to date)

Read: https://cwe.mitre.org/data/definitions/367.html

9.1 Time-of-Check Time-of-Use (TOCTOU) Race Condition

Race Condition and TOCTOU

- In general, a ***race condition*** occurs when **multiple processes** access a piece of **shared data** in such a way that the **final outcome** depend on the sequence of accesses

- In the context of security, the "**multiple processes**" typically refer to:
  1. A (**vulnerable**) process ***A*** that **checks/verifies** the permission to access a shared data, and subsequently **accesses** the data
  2. Another (**malicious**) process ***B*** that "**swaps**" the data

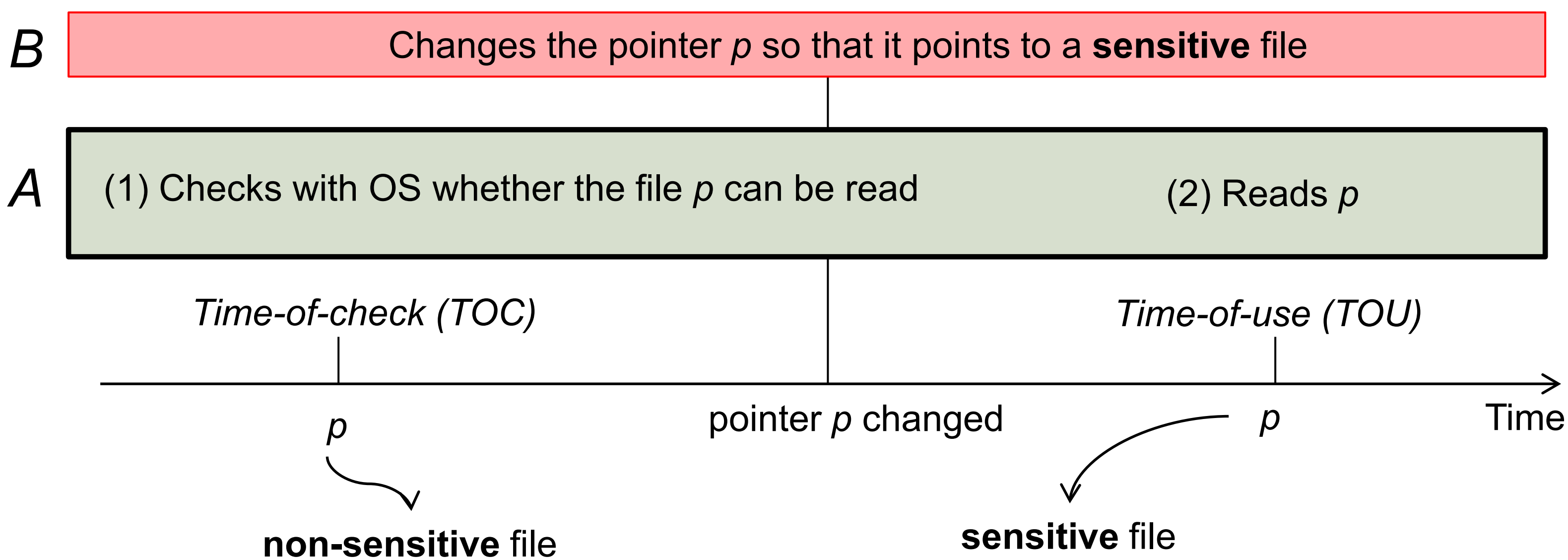
- Note that the two processes can be run by ***one*** malicious user in the system

Race Condition and TOCTOU

- There is a "**race**" between processes ***A*** and ***B***:

  If ***B*** manages to complete the swapping before ***A*** accesses the data, then the attack succeeds

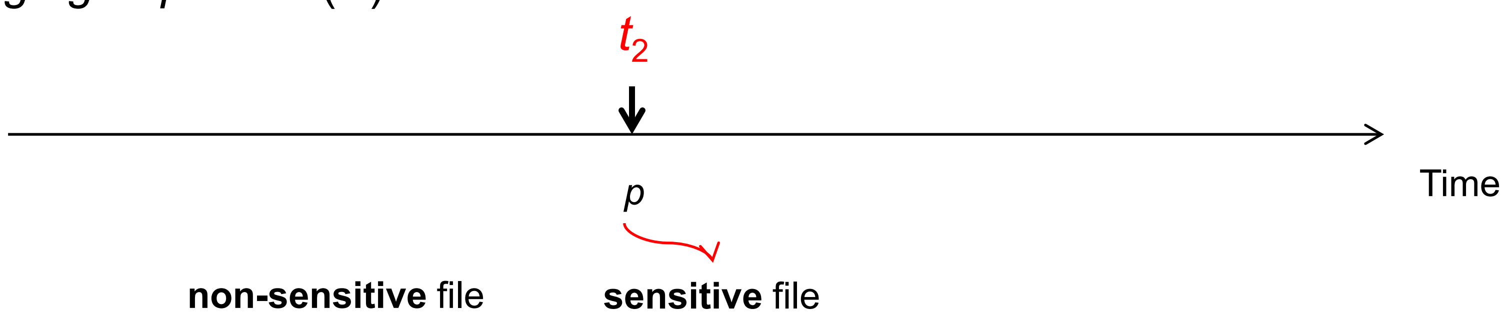
- This scenario is also known as ***Time-of-check time-of-use*** **(TOCTOU)**



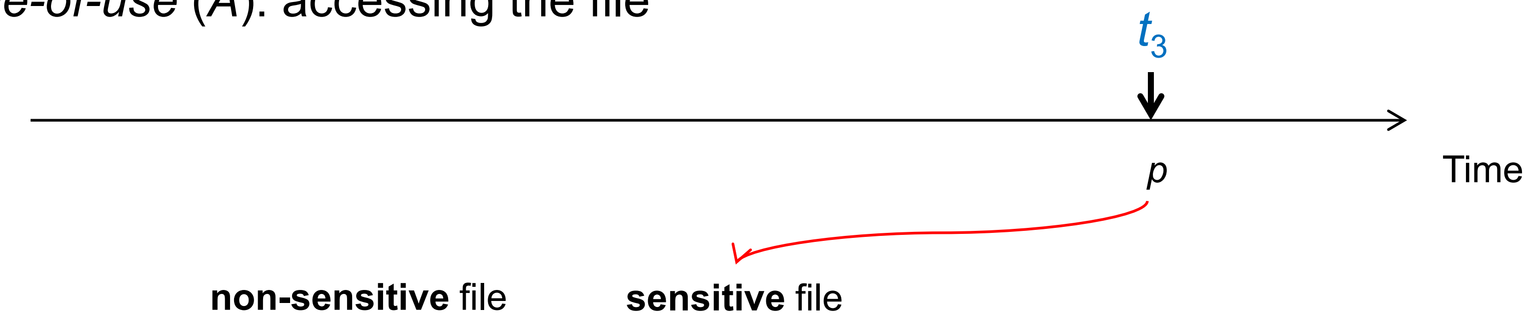
TOCTOU: Three Important Events/Actions

time-of-check (*A*): checking whether the process is authorized to access the file

changing of pointer (*B*)



time-of-use (*A*): accessing the file



A Sample Case: The Vulnerable Program

- A variant of Example 2 in https://cwe.mitre.org/data/definitions/367.html is shown next
- This setUID program is to be ran under an **elevated privilege** (i.e. setUID-root)
- The **`access(f,W_OK)`** system call:
  - **Checks** whether the user executing the program has the permission to **write** to the specified *filename* **`f`**
  - Returns **0** if the process **has** the permission
  - The check is done based on the process's ***real UID***
- Because of the check on real UID, a malicious user needs to find **a way** to access a sensitive target file

A Sample Case: The Vulnerable Program (Continued)

TOC

```
//   f is a string that contains the name of a file
//   fd is the file descriptor

if(!access(f, W_OK))        // check whether the real UID has write permission to f
{
    fprintf (stderr, "permission to operate %s granted\n", f );
    fd = open(f,O_RDWR);    // open the file with read and write access
    OP(fd);                 // a routine that operates on the file. OP is not a system call
    ....
}
else
{
    fprintf(stderr,"Unable to open file %s.\n", f );
}
```


TOU

access() System Call

ACCESS(2) Linux Programmer's Manual ACCESS(2)

NAME

access, faccessat - check user's permissions for a file

SYNOPSIS

#include <unistd.h>

int access(const char *pathname, int mode);

...

DESCRIPTION

access() checks **whether the calling process can access the file pathname**. **If pathname is a symbolic link, it is dereferenced**.

The mode specifies the accessibility check(s) to be performed, and is either the value F_OK, or a mask consisting of the bitwise OR of one or more of R_OK, W_OK, and X_OK. F_OK tests for the existence of the file. **R_OK, W_OK, and X_OK test whether the file exists and grants read, write, and execute permissions, respectively.**

The check is done using the calling process's real UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., open(2)) on the file. Similarly, for the root user, the check uses the set of permitted capabilities rather than the set of effective capabilities; and for non-root users, the check uses an empty set of capabilities.

...

A Sample Case: Two Files Involved

- Suppose the program has the "setUID-root" permission, and `bob` (a malicious user) invokes the program
- Thus the process' **real UID** is **`bob`**, but its **effective UID** is **`root`**

- Furthermore, suppose that the filename `f` is

  `/usr/year1/bob/list.txt`

  which is a file **owned by `bob`**: `bob` has a permission to **change** it

- After `bob` has started the process, he immediately **replaces** the file

  `/usr/year1/bob/list.txt`

  with **a symbolic link** that points to a **sensitive system file**

- `bob` **does not** have a write permission to the sensitive system file, but he wants to change the file: *Can he do that?*

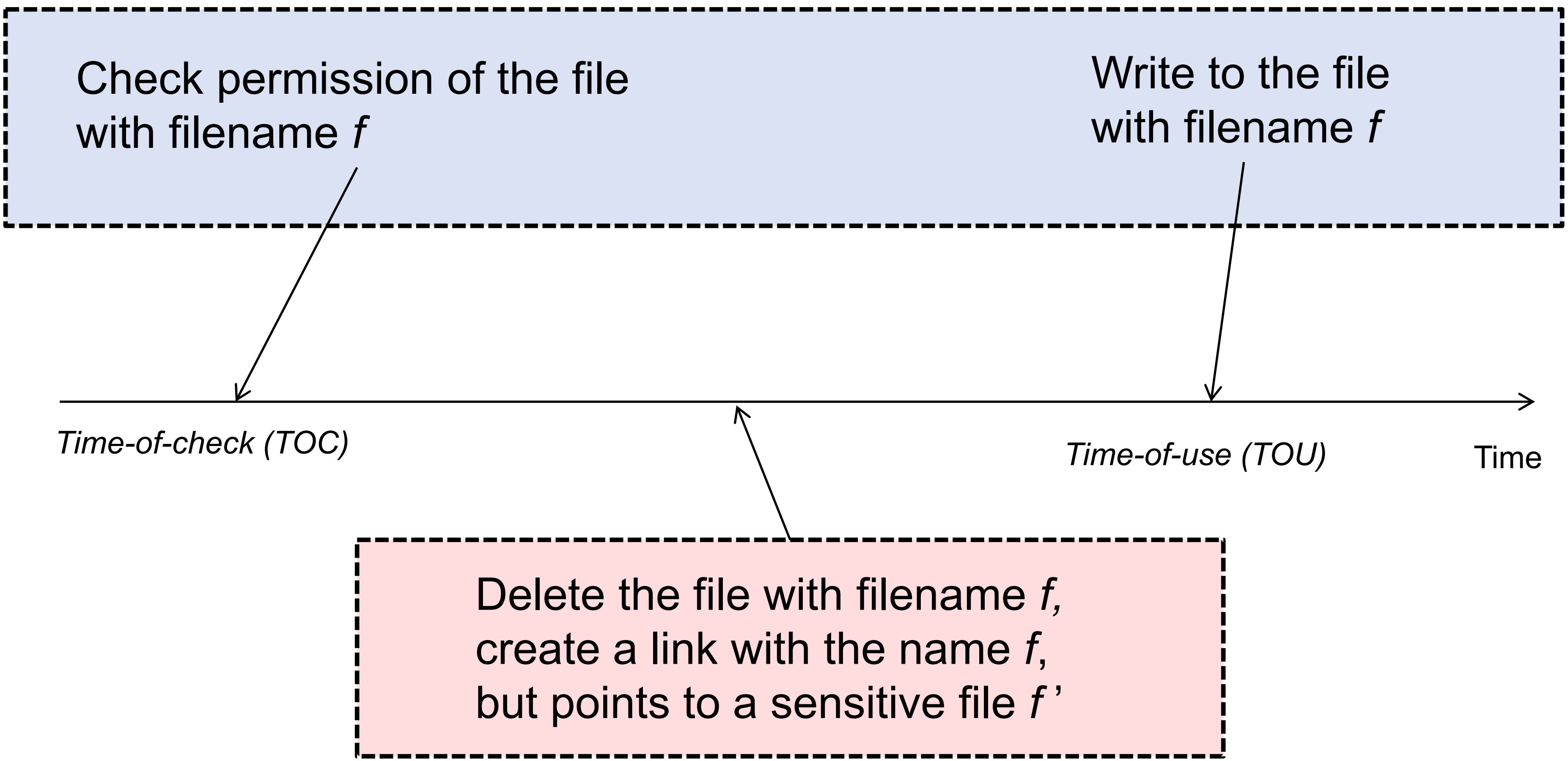
A Sample Case: Bob's Malicious Actions

- This can be done by running **a script** that carries out the following 2 steps:
  1. **Delete** `/usr/year1/bob/list.txt`
  2. Create **a symbolic link** with the filename `f` and points to a system file, by issuing this Unix command:

     ```
     ln -s /usr/course/cz2017/grade.txt
           /usr/year1/bob/list.txt
     ```

- With ***some probability***, `bob` **wins** the race
- Hence the process operates on the system file `/usr/course/cz2017/grade.txt` instead of `/usr/year1/bob/list.txt`

A Sample Case: Timeline of Events



`/usr/year1/bob/list.txt` `/usr/course/cz2017/grade.txt`

Avoiding TOCTOU on File Checking (C Programming): Approach 1

Defense approach 1:

- In your program, **avoid** using **separate system calls** that takes the **same filename** as input
- Instead, try to **open** the file **only once** (and thus ***lock*** it to ***block*** any further changes by other processes), and use the file handle/descriptor

- That is, in general, **avoid** using **`access()`** system call
- Rather, open the file once using **`open()`** system call, and then use **`fstat()`** system call to check the permission

Safe Versions: Approach 1

TOC

TOU

```
//  f is a string that contains the name of a file

fd = open(f, O_RDWR);       // open the file with read and write access

fstat(fd, &filestatus) )   // get the file status and store them to filestatus
if ( CP(filestatus) )      // check permission of the file (CP is not a system call)
{
    fprintf (stderr, "permission to operate %s granted\n", f );
    OP(fd);            // a routine that operates the file (OP is not a system call)
    ....
}
else
{
    fprintf(stderr,"Unable to open file %s.\n", f );
}
```

Avoiding TOCTOU on File Checking (C Programming): Approach 2

Defense approach 2:

- A **better practice** would be:
  avoid writing your own access-control on files,
  and leave the checking to **the underlying OS**
  after appropriately setting your process credentials

- For instance, adopt the following in your program:
  set the process' **effective UID** to the **appropriate**
  (normal/non-root) user before **accessing** the file

- In this way**, the OS** checks the permission,
  and decide whether to grant or deny the access

- After it is done, **reset** the **effective UID** back to root
  if required

Safe Versions: Approach 2

Elevated Privilege

Lowered privilege

Elevated (root) privilege restored

```
//  f is a string that contains the name of a file
//  u is the UID of the appropriate user (i.e. Alice)
//  r is the UID of root;
…

seteuid (u);                    // from now onward, the effective UID is u
fd = open (f, O_RDWR);
OP (fd);
…

seteuid (r);                    //  from now onward, the effective UID is root

…
```

9.2 Defense and Preventive Measures

Read http://en.wikipedia.org/wiki/Bounds_checking

General Comments

- As illustrated in previous examples, **many bugs** and **vulnerabilities** are due to **programmer's ignorance**
- In general, it is **difficult** to analysis a program to ensure that it is bug-free (recall the "halting-problem")
- There is no "fool-proof" method
- However, various useful **counter measures** are available as discussed next

9.2.1 Input Validation using Filtering

Filtering

- In almost all examples (except TOCTOU) we have seen, the attack is carried out by feeding **carefully-crafted input** to the system
- Those inputs **do not** follow the **"expected" format**: e.g. the input is too long, contains control/meta characters, contains negative number, etc.
- Hence, a preventive measure is to perform an **input validation/filtering** whenever an input is obtained from the user: if the input is not of the expected format, reject it

Problems with Filtering

- It is *difficult* to ensure that the filtering is **"complete"** (i.e. it doesn't miss out any malicious strings), as illustrated in the example on UTF
- In that example on UTF, the input validation intend to detect the substring " . . / "
- Unfortunately, there are **multiple representations** of " . . / " that the programmer is not aware of
- A filter that completely **blocks all bad inputs** and **accepts all legitimate inputs** is *very difficult* to design

Filtering

- There are generally **two approaches** of filtering:
  1. ***White list***: A list of items that are known to be **benign** and allowed to **pass**, which could be expressed using regular expression
     However, some legitimate inputs may be blocked.
  2. ***Black list***: A list of items that are known to be **bad** and to be **rejected**.
     For example, to prevent SQL injection,
     if the input contains meta characters, reject it.
     However, some malicious input may be passed.
- *Which type of filtering is then more secure?*

9.2.2 Using “Safer” Functions

Safer Function Alternatives

- Completely **avoid** functions that are known to create problems
- Use the **“safer” versions** of the functions
- C/C++ have many of those:

  `strcpy()` ←→ `strncpy()`

  `printf(`*f*`)`

  `access()`

- Again, even if they are avoided, there ***could*** still be vulnerability: recall the example that uses a combination of `strlen()` and `strncpy()` in your **tutorial**

9.2.3 Bounds Checking and Type Safety

Bounds Checking

- Some programming languages (e.g. Java, Pascal) perform ***bounds checking*** at runtime
- That is, when an array is declared, its upper and lower bounds have to be declared
- At runtime, whenever a reference to an array location is made, the index/subscript is **checked** against the upper and lower bounds
- Hence, a simple assignment like:

```
a[i] = 5;
```

would **consists of** these steps:

1. Checks that `i` is >= the lower bound;
2. Checks that `i` is <= the upper bound; and
3. Assigns 5 to the memory location

Bounds Checking

- If the checks **fail**, the process will be **halted**
  (or an exception is to be thrown as in Java)
- The added first 2 steps reduce efficiency,
  but will **prevent** buffer overflow
- Many of the known vulnerabilities is due to buffer overflow that can be prevented by this **simple bounds checking**: visit http://cve.mitre.org/cve/cve.html to see how many entries contains “buffer overflow” as keywords

- The infamous C and C++ **do not** perform bounds checking!
- Yet, **many** pieces of software are written in C/C++!

Some Words of Wisdom

C. A. R. Hoare (1980 Turing Award winner) on his experience in the design of ALGOL 60, a language that **included** bounds checking:

> "A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interest of efficiency on production runs. Unanimously, they urged us not to—they already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. **In any respectable branch of engineering, failure to observe such elementary precautions *would have long been against the law*.**"

Type Safety

- Some programming languages carry out ***"type" checking*** to ensure that the arguments an operation get during execution are always correct
- For example: **`a = b;`**
  if `a` is a 8-bit integer, `b` is a 64-bit integer, then the type is **wrong**
- The checking could be done at **runtime** (i.e. **dynamic type checking**), or when the program is being **compiled** (i.e. **static type checking**)
- Bounds checking can also be **considered** as one mechanism that ensures "type safety"
- For example in Pascal:

```
Type    indexrange = 1..10;
Var     A: array [indexrange] of integer;
Begin
    A[0] = 5;
```

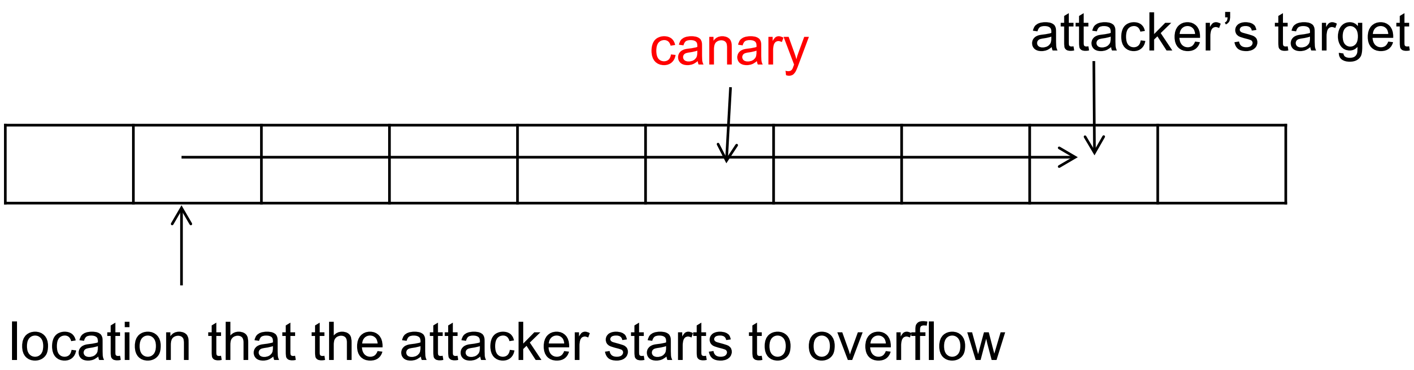
wrong type

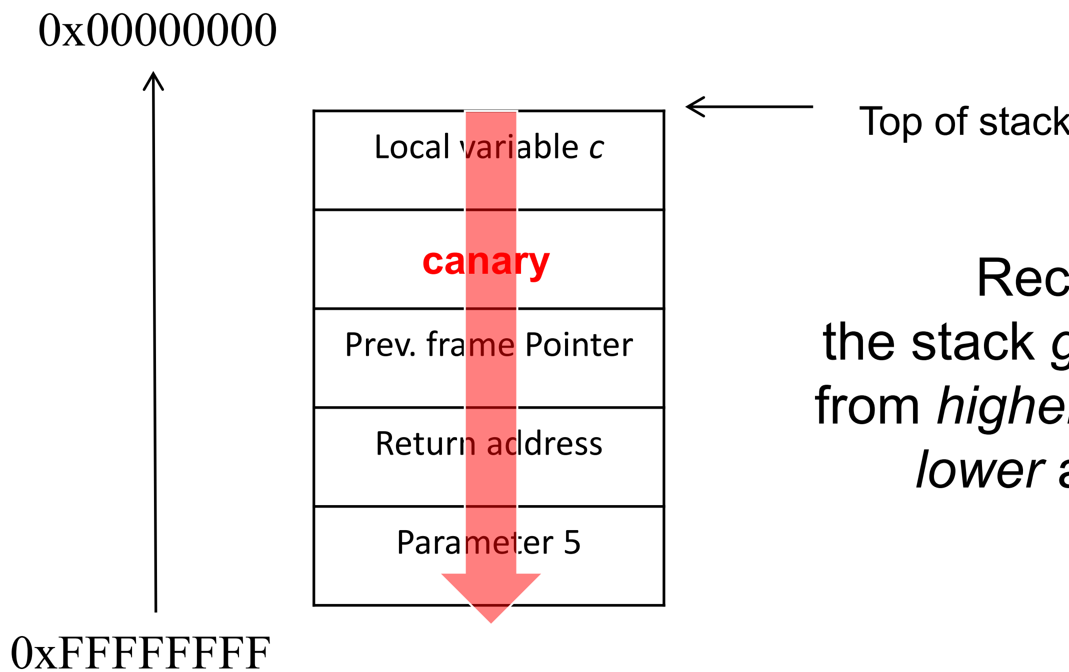
9.2.4 Memory Protection (Randomization, Canaries)

Canaries

- **Canaries** are "secret" values inserted at carefully selected memory locations at runtime
- Checks are carried out at **runtime** to make sure that the values are **not** being modified: if so, halts
- Canaries can help ***detect*** overflow, especially stack overflow:
  - In a typical buffer overflow, **consecutive** memory locations have to be over-ran: the canaries would be **modified**
- It is important to keep the values "**secret**":
  if the attacker knows the value, it may able to write the secret value to the canary while over-running it!
- (***Optional***) In Linux, you can **turn off** gcc canary-based stack protection by supplying this flag when invoking gcc:
  `-fno-stack-protector`

Canaries





Recall that:
the stack *grows upward*,
from *higher* addresses to
lower addresses

Stack Smashing Detected: Program 1

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

int main(int argc, char *argv[])
{
    char text[15];

    strcpy(text, argv[1]);
    printf("Your supplied argument is :%s\n", text);

    printf("main() is exiting\n");
    return 0;
}
```

Stack Smashing Detected: Program 1

- If compiled **with** stack protector:

```
./program-wsp aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
main() is exiting
*** stack smashing detected ***: ./program-wsp
terminated
Aborted (core dumped)
```

- If compiled **without** stack protector:

```
./program-wosp aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
main() is exiting
Segmentation fault (core dumped)
```

Stack Smashing Detected: Program 2

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

void copy_this(char *arg1)
{
   char text[15];

   strcpy(text, arg1);
   printf("Your supplied argument is :%s\n", text);
   printf("Function is exiting\n");
}

int main(int argc, char *argv[])
{
    copy_this(argv[1]);

    printf("main() is exiting\n");
    return 0;
}
```

Stack Smashing Detected: Program 2

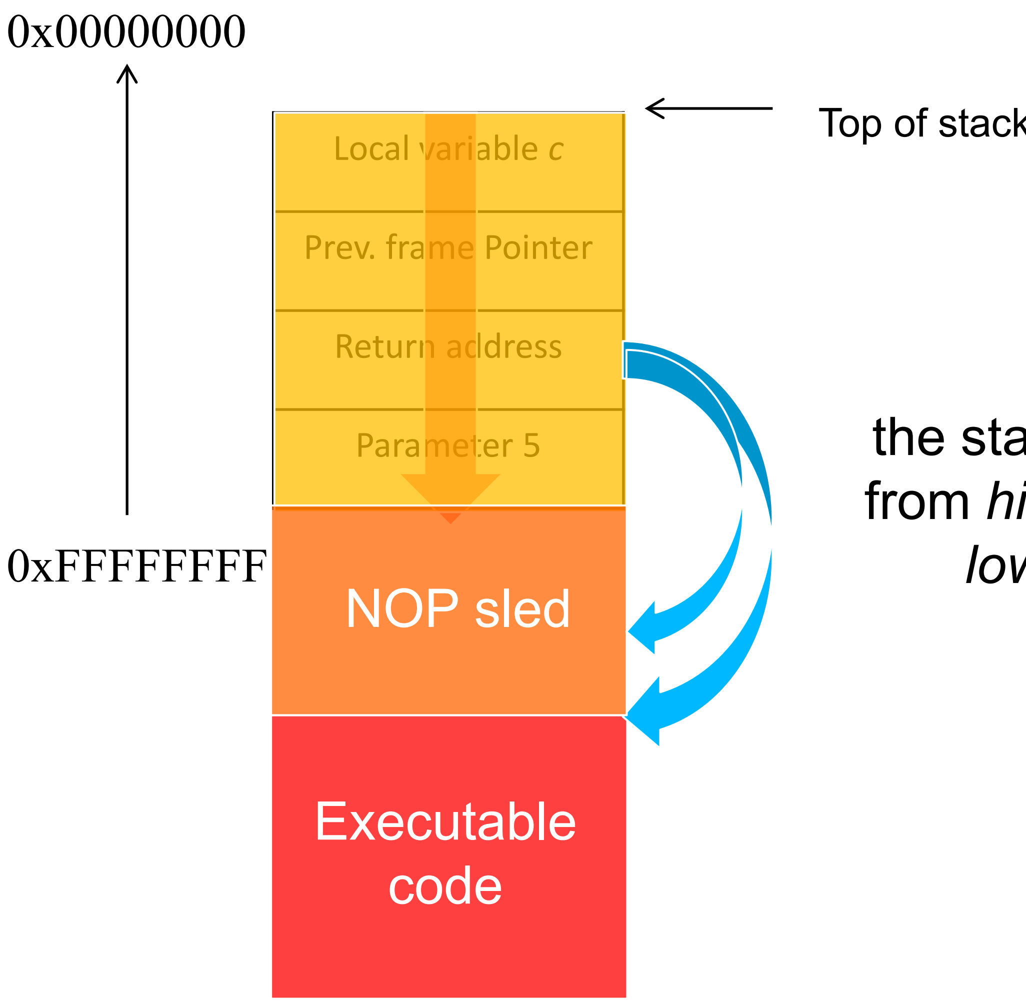
- If compiled with stack protector:

```
./program-too-wsp
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Function is exiting
*** stack smashing detected ***: ./program-too-wsp
terminated
Aborted (core dumped)
```

- If compiled *without* stack protector:

```
./program-too-wosp
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Function is exiting
Segmentation fault (core dumped)
```

Stack Smashing (Stack Overflow): Illustration



Recall that:
the stack *grows upward*,
from *higher* addresses to
lower addresses

Memory Randomization

- It is to the attacker's advantage when the data and codes are always stored in the same locations in memory
- ***Address space layout randomization (ASLR)*** is a **prevention** technique that can help decrease the attacker's chance
- ASLR: **randomly** arranges the address space positions of **key data areas** of a process, including:
  the base of the executable and the positions of the stack, heap and libraries
- *(Details are omitted in this module)*
- ***Optional***: in Linux, you can turn off (disable) address randomization using:
  ```
  sudo sysctl -w kernel.randomize_va_space=0
  ```

9.2.5 Code Inspection

Code Inspection

- **Manual checking**: manually checks the program, is tedious
- **Automated checking**: some automations and tools are possible
- An example is ***taint analysis***:
  - Variables that contain input from the (potential malicious) users are labeled as ***sources***
  - Critical functions are labeled as ***sinks***
  - Taint analysis **checks** whether any of the sink's arguments could potentially be affected (i.e. tainted) by a source
  - Example: sources = user input
    sink: opening of system files, function evaluating a SQL command
  - If so, special check (e.g. manual inspection) would be carried out
  - Taint analysis can be **static** (i.e. checking the code without running/ tracing it), or **dynamic** (i.e. running the code with some inputs)

9.2.6 Testing

Testing

- Vulnerability can be discovered via **testing**
- Types of testing:
  - ***White-box*** testing:
    the tester has an access to the application's **source code**
  - ***Black-box*** testing:
    the tester does not have an access to the source code
  - ***Grey-box*** testing:
    A combination of the above, reverse-engineered binary/executable

Testing

- Security testing attempts to discover **intentional attack**, and hence would test for inputs that are **rarely occurred** under normal circumstances
- Examples: very long names, names containing numeric values, string containing meta characters, etc.
- ***Fuzzing*** is a technique that sends **malformed inputs** to discover vulnerabilities:
  - There are techniques that are more effective than sending in random inputs
  - Fuzzing can be automated or semi-automated: *(the details are not required)*
- **Terminology**: white list vs black list, white-box testing vs black-box testing, white hat vs black hat

9.2.7 The Principle of Least Privilege

The Principle of Least Privilege

Apply the "***Principle of Least Privilege***":

- When writing a program, be **conservative** in elevating the privilege
- When deploying software system, do **not** give the users more access rights than necessary, and do **not** activate unnecessary options

The Principle of Least Privilege

- Example:

  Software contain many **features**: e.g. a web-cam software could provide many features so that the user can remotely control it.
  A user can choose to set which features to be on/off.

  Suppose you are the developer of the software.
  Should all features to be **switched on by default** when the software is shipped to your clients?

  If so, it is **the clients' responsibility** to "**harden**" the system by selectively **switch off the unnecessary features**.
  Your clients might not aware of the implications and thus at a higher risk.

- Terminology: What does "*hardening*" mean?

9.2.8 Patching (Keeping up to Date)

Vulnerability Lifecycle

- **Life-cycle of a vulnerability**:
  (1) a **vulnerability** is discovered → (2) affected code is fixed →
  (3) the revised version is tested → (4) a **patch** is made public →
  (5) patch is applied
- In some cases, the vulnerability could be announced (1?) **without the technical details** before a patch is released: the vulnerability is likely to be known to only a small number of attackers (even none) before it is announced
- When **a patch** is released (4), the patch can be **useful to attackers** too: they can inspect the patch and derive the vulnerability
- Hence, interestingly, the number of successful attacks can **go up** after the vulnerability/patch is announced: since more attackers would be aware of the exploit (see the next slide)

Vulnerability Lifecycle

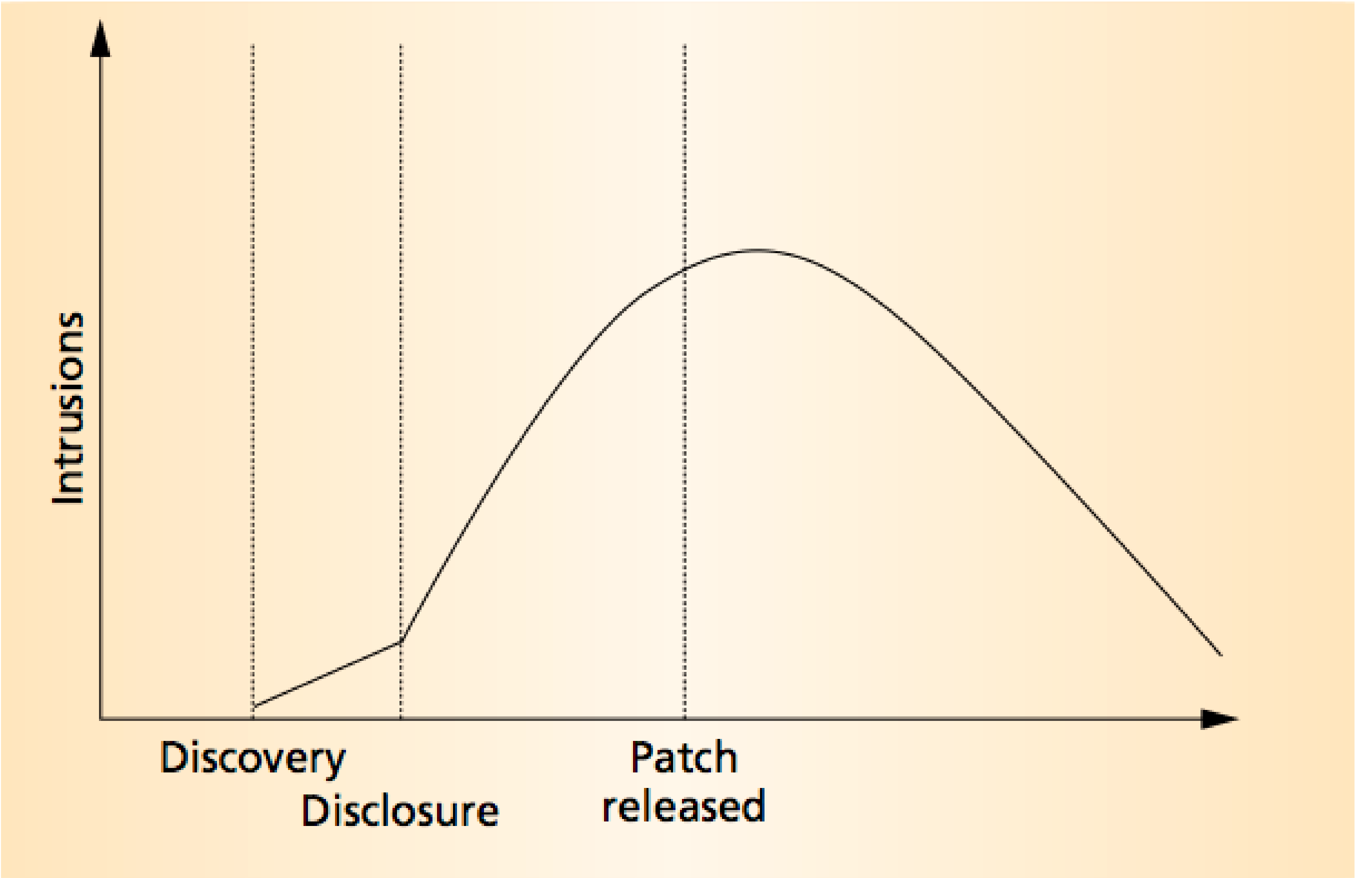


Figure 1. Intuitive life cycle of a system-security vulnerability. Intrusions increase once users discover a vulnerability, and the rate continues to increase until the system administrator releases a patch or workaround.

image obtained from
William A. Arbaugh et al. Windows of vulnerability: A case study analysis. IEEE Computer, 2000.

http://www.cs.umd.edu/~waa/pubs/Windows_of_Vulnerability.pdf

Patching

- It is crucial to apply the patch **timely**
- Although it seems easy, applying patches is **not** that straightforward:
  - For **critical systems**, it is not wise to apply the patch immediately before rigorous testing:
    E.g. after a patch is applied, the train scheduling software crashes
  - Patches might **affect** the applications, and thus affect an organization **operation***:*
    E.g. after a student applied a patch on Adobe Flash, he couldn't upload a report to LumiNUS and thus missed a project deadline
- "**Patch management**" is a field of study
- See the guide on patch management issued by NIST:
  "*Guide to Enterprise Patch Management Technologies*", 2013,
  http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-40r3.pdf

Summary: Defense Mechanisms and Stages of SDLC

Adopt various counter measures at **different stages of SDLC**:

- **Development** stage:
  - Using safer functions
  - Bounds checking and type safety
  - Filtering (input validation)
  - Code inspection (taint analysis)
  - The principle of least privilege*
  - Executable generation with enabled memory protection*
- **Testing** stage:
  - Testing: fuzzing, penetration testing
- **Deployment** (including software execution) stage:
  - Runtime memory protection*: randomization
  - The principle of least privilege*: disable unnecessary features
  - Patching

* Applicable to *multiple* stages