language=Java, basicstyle= , keywordstyle=, stringstyle=, commentstyle=, morecomment=[s][]/***/, tabsize=2, showspaces=false, showstringspaces=false, texcl = true, rulecolor = , breaklines = true, aboveskip = 0em, belowskip = 0em

1 Secure Software Principles

Three pillars
- Risk Management
- Touchpoints
- Knowledge

1.1 80% / 20% Problem Reduction

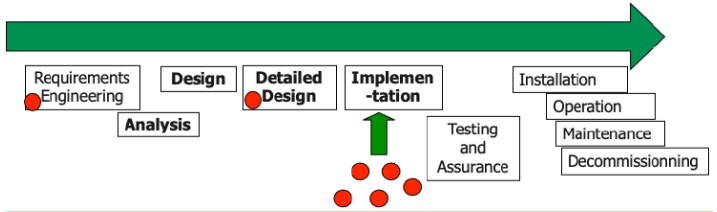
1. Secure the weakest link
2. Practice defence in depth
3. Fail securely
4. Follow the principle of least privilege
5. Compartmentalize
6. Keep it simple
7. Promote privacy
8. Remember that hiding secrets is hard
9. Be reluctant to trust (Off-the-shelf software)
10. Use your community resources (crypto algorithms)

2 Secure Software Lifecycle

SW Engineering processes: Waterfall, Spiral, Prototyping, XP, Adaptive Programming, ...
None of these really support security.

2.1 Software Engineering & Security

- Often ad hoc or not covered at all
- Issues:
  - Non systematic approach
  - Bad early decisions
  - Hard to manage and modify



2.2 Requirements / Analysis

- Identify security requirements
- Quantify the security risks
- Know the entire problem domain of the system

2.2.1 Identify Security Requirements

1. Identify stakeholders
2. Identify assets from different stakeholders (sensitive info / resources)
3. Identify requirements on these assets

Result: high-level security policy
STRIDE: Systematic approach for threat identification.
- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation disclosure
- **D**enial of service
- **E**levation of privilege

2.2.2 Quantify Security Risks

- Estimate and quantify the risk of the previously identified threats
  1. Importance or severity (critical vs. non-critical)
  2. Cost
  3. DREAD (Damage potential, Reproduceability, Exploitability, Affected users, Discoverability)
- Order the requirements and identify the relevant subset: Risk mitigation strategy

Result: high-level security policy revisited

2.2.3 Example

General formulated:
The software must validate all user input to ensure it does not exceed the size specified fot that type of input.
The system must encrypt sensitive data transmitted over the Internet between the server and the browser.

Non-functional requirements:

1. **Security Property Requirement**: specifies the characteristics that software must exhibit.
2. **Constraint or Negative Requirements** limit what software functionality can be allowed to behave.
3. **Security Assurance Requirements** are rules or best practices by which the software security functions will be built, deployed and operated.

2.3 Detailed Design

Main goals:

1. Identify security technologies that meet relevant security requirements
2. Bind these technologies to the application

2.3.1 Identify security technologies

- Select requirements that should be addressed at this level
- Identify security technologies that address requirements

STRIDE Countermeasures: Tampering
- Use data hashing and signing
- Digital signatures
- Strong authorization
- Tamper-resistant protocols across communication links
- Secure communication links with protocols that provide message integrity

Information Disclosure
- Strong authorization
- Strong encryption
- Secure communication links with protocols that provide message confidentiality
- Do not store secrets in plaintext

...

2.3.2 Bind Technologies to Application

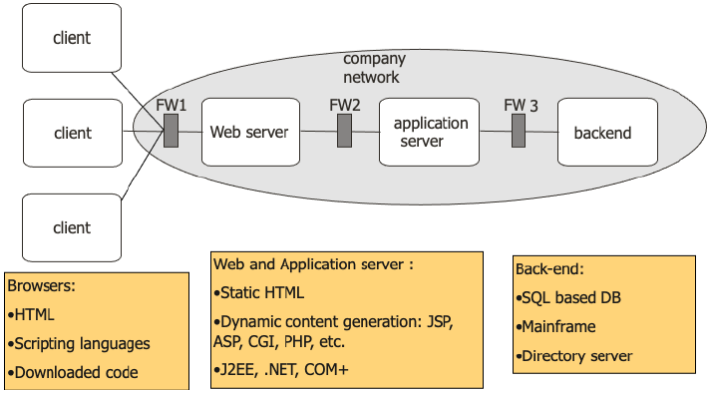
- For coarse-grained or simple requirements
  - Countermeasure can be realized in the operating system/middleware (SSL)
- For fine-grained and complex requirements
  - Implement as part of the application (hard to get right)

2.4 Other phases

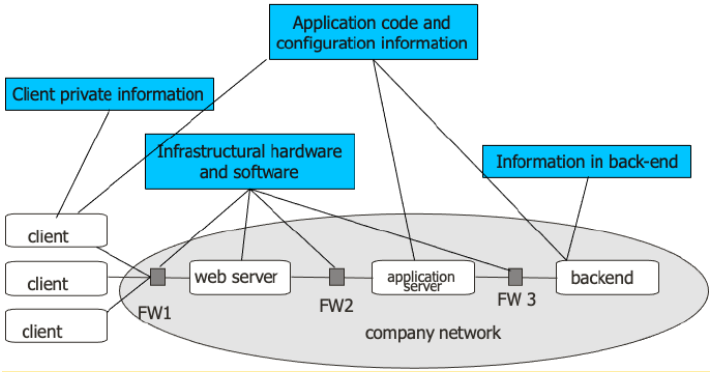
- Avoiding implementation vulnerabilities
- Security testing
- Automated patching

2.5 Case study: Web Applications

2.5.1 High level Architecture



2.5.2 Owners and Assets



2.5.3 Threat Agents and Threats

Any hacker on the internet:
- Spoof client or server
- Eavesdrop on connections / modify data in transit
- Bring down infrastructure components
- Gain unauthorized access to application or data

Authorized user of the application: (additionally)
- Repudiate transactions
- Elevate privilege

Malicious server:
- Steal private client information
- Spread spyware/viruses

2.5.4 Infrastructural countermeasures

Authentication:
- Network level: IPSEC
- Transport level: HTTP authentication mechanisms
- OS level: Windows authentication
- Single sing-on systems based on federation or windows active directory (Kerberos)
- Web authentication products: IAM products

Data protection:
- IPSEC (Network)
- Kerberos (OS)
- TLS (Transport)

Access control:
- Firewall
- In the webserver (URL)
- RBAC in the web container or application server
- File system based access control
- Access control products

Sandboxing:
- At the OS level: low-privileged accounts
- Language level: Java security architectures

Others:
- Filtering
- Throttling
- Shielding
- ...

2.5.5 Typical vulnerabilities

Bugs in functional parts
- Input validation
- Race conditions
- Bad error handling

Broken countermeasures
- Access control
- Authentication
- Crypto

2.5.6 Available Countermeasures at coding level

- Security technologies
- Quality improvements
  - Choice of programming language
  - Coding guidelines

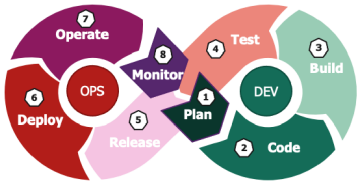
- Source code scanners
- Security testing and audit
- Code review

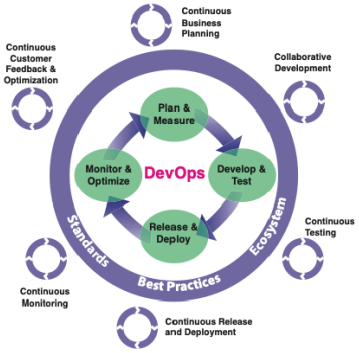
2.6 Daily Sins

1. Buffer Overruns
2. Format String problems
3. Integer Overflows
4. SQL Injection
5. Command Injection
6. Failing to Handle Errors
7. XSS
8. Failing to protect Network traffic
9. Use of magic URLs and Hidden Forms
10. Improper use of SSL and TLS
11. Use of Weak Password-Based systems
12. Failing to store and protect data securely
13. Information leakage
14. Improper file access
15. Trusting Network Name Resolution
16. Race Conditions
17. Unauthenticated Key Exchange
18. Cryptographically strong random numbers
19. Poor usability

3 Threat Modeling

3.1 Securing DevOps





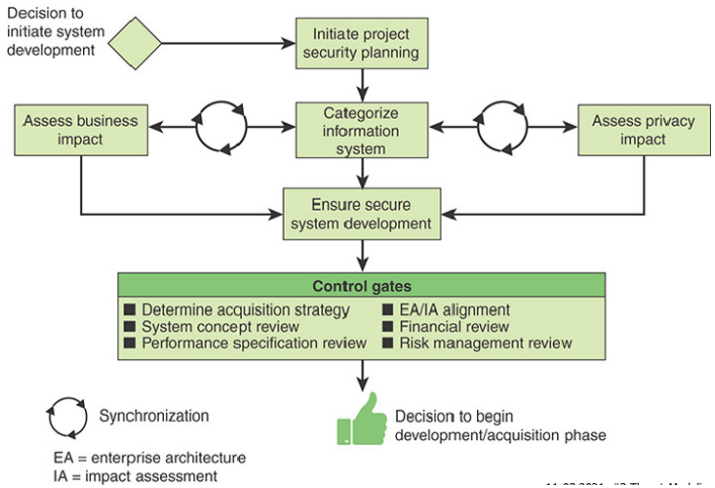
3.2 Four Major Activities

- Plan and measure
- Develop and test
- Release and deploy
- Monitor and optimize

3.3 Continous Secutity

- Prepare the organisation
- Protect the software
- Produce well secured software
- Respond to vulnerabilities

3.4 Security in the Phases



Major Security Activities: A number of security-related activities are needed to assure that security is incorporated effectively in that design phase
Expected Outputs: A key to success is to define specific deliverables for each activity
Synchronization: A feedback loop between tasks provides opportunities to ensure that the SDLC is implemented as a flexible approach that allows for appropriate and consistent communication and the adaptation of tasks and deliverables as the system is developed
Control gates: Decision points at the end of each phase when the system is evaluated and management determines whether the project should continue as is, change direction, or be discontinued

3.4.1 Initiate project security planning

- Identify key security roles
- Identify the standards and regulations for the system
- Develop an overall plan for security milestones
- Get Stakeholders to have a common understanding (security implications, considerations, requirements)
- Enable developers to design security features
- **Output:** Supporting documents of all decisions

3.4.2 Categorize information system

- Identify information that will be transmitted, processed or stored
- Define applicable levels of information categorization (based on impact analysis)
- **Result:** Catalog of information types
- **Outputs:** Definitions of categories, level of effort estimate

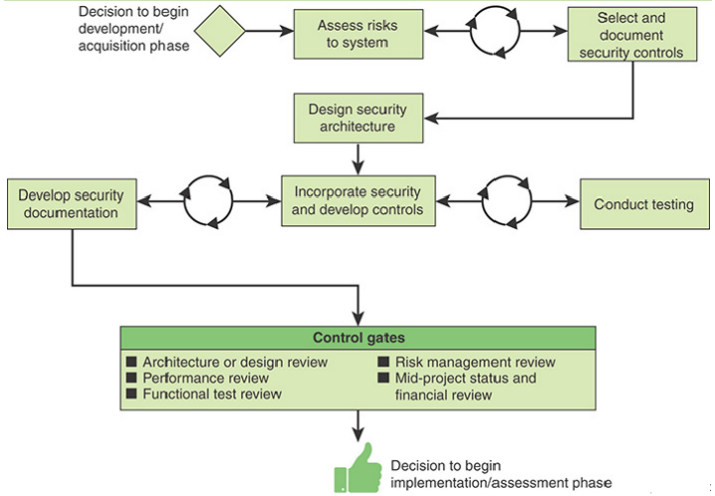
3.4.3 Ensuring secure system development

- Develop a set of principles of security expectations
  - Secure concept of operations
  - Standards and processes
  - Security training for development team
  - Quality management
  - Secure environment
  - Secure code practices and repositories
- **Output:** Plans for development security training / quality assurance

3.4.4 Control Gates

- Determine acquisition strategy
- System concept review
- Performance specification review
- EA/IA alignment
- Financial review
- Risk management review

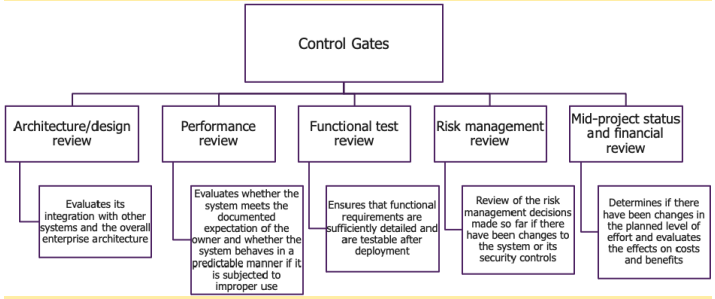
3.5 Develop & Test



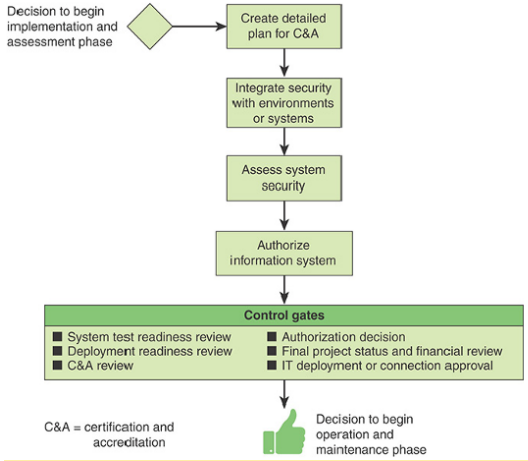
3.5.1 Designing the security architecture

- Produce a detailed architecture
- Incorporate security features and controls into the system design
- **Outputs:**
  - Schematic of security integration
  - List of shared services
  - Identification of common controls used by the system

3.5.2 Control Gates



3.5.3 Begin Implementation / Assessment phase

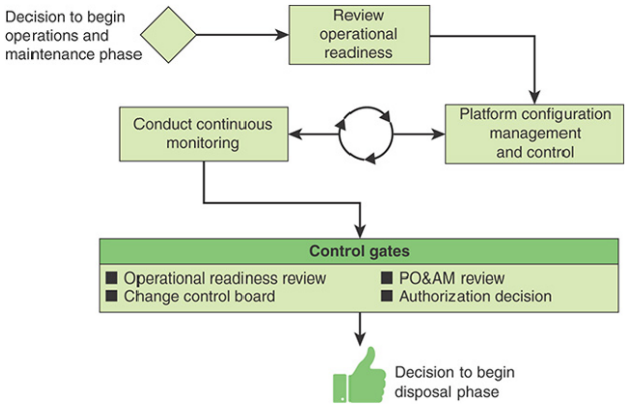


3.5.4 Control Gates

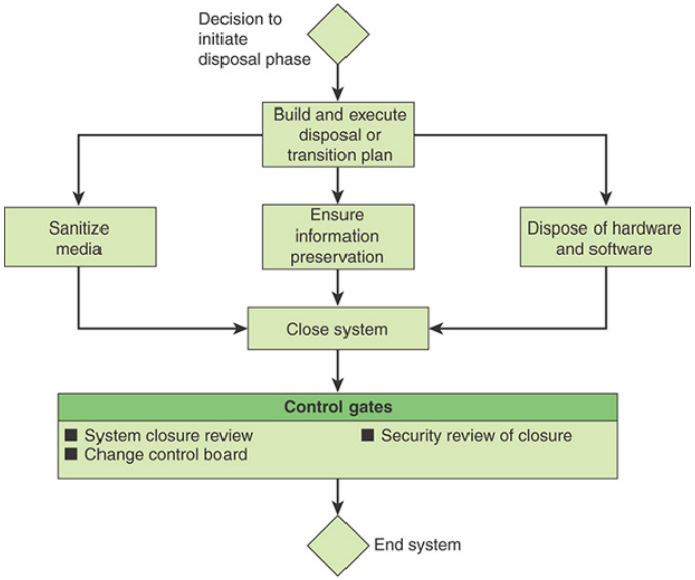
- System test readiness review

- Deployment readiness review
- Certification and accreditation review
- Authorization decision
- Final project status and financial review
- IT deployment or connection approval

3.5.5 Begin Operations and Maintenance phase



3.5.6 Initiate disposal phase



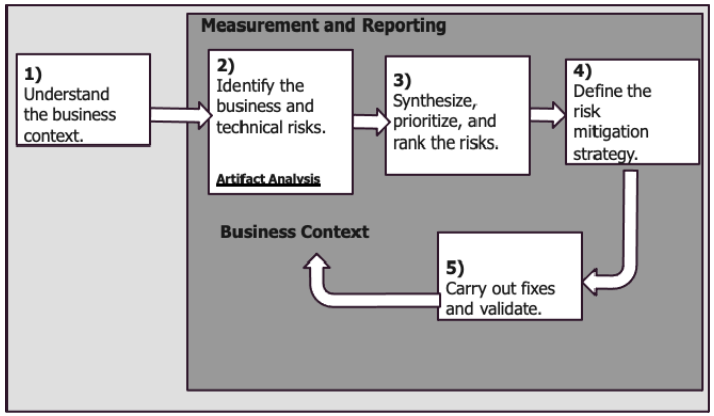
3.6 Continous Security

1. Test driven Security
2. Monitor and respond to attacks
3. Assess risks and mature security

3.7 Risk Management

3.7.1 Activities

Summary:



1. Understand the business context
   - Extract and describe business goals
   - Set prios
   - Understanding what risks to consider
   - Gathering the artifacts
   - Conducting project research to the scope
2. Identifying the business and technical risks (priorize / rank)
   - Business risks impact business goals
   - Map technical risks to business goals
   - Develop a set of risk questionnaires
   - Interview the target project team
   - Analyse the research interview data
   - Evaluate software artifacts
3. Synthesize, prioritize and rank the risks
   - Prio the risks based on business goals
   - Apply risk metrics
   - Number of risks emerging over time
   - What shall be done first?
   - What is the best allocation of resources?
4. Define the risk mitigation strategy
   - Take into account: Cost, Implementation time, Likelihood of success, Competence and impact
   - Identify the validation techniques
   - Metrics are financial in nature
5. Carry out fixes and validate their correctness
   - Implement the mitigation strategy
   - Rectify artifacts

3.7.2 Summary

- Relies on continuous and consistent identification of risks.
- The five fundamentals should be applied repeatedly
- Use project management tools to track risk information

3.8 Attack Trees

- Tree-structured graph
- Showing how a system can be attacked

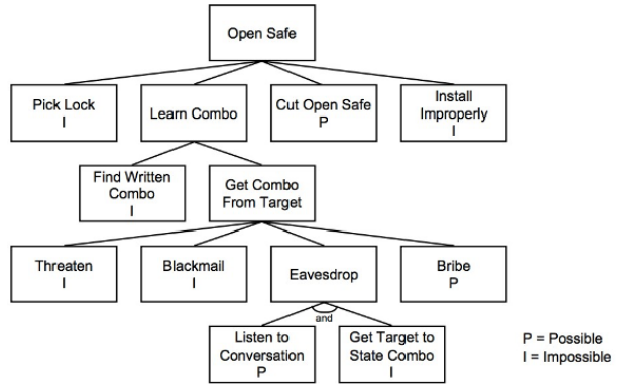
Construction:

1. Identify goals (1 Tree per goal)
2. Identify attacks against goals
3. Existing sub-trees can be plugged in

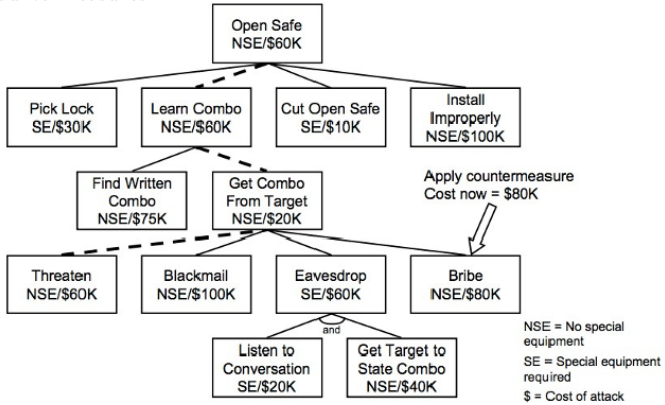
Usage:

- Propagate up the tree
- You can specify values that represent other different meanings (Equipment / Cost)
- Combine Node Values

3.8.1 Example



Countermeasures:



4 Threats & Attacks

4.1 Threat Risk Modeling

1. Identify security objectives with a focus on:
   - sensitive information stored on device
   - third party libraries
   - loss of reputation derived from misuse of the app
2. Break down app features, identify security impact
3. Identification of related threats and vulnerabilities

4.2 Process Steps

1. Identify assets
2. Create an architecture overview (simple diagrams)
3. Decompose the application
4. Identify the threats
5. Document the threats
6. Rate the threats

4.3 Threat Modeling - WHY?

For developers

- When used in dev cycle, exposure to security risks can be minimized
- Identifies, prioritize abd categorize the threads found making issues that are easily manageable

For Penetration Testers

- Initial Analysis of the architecture
- Common languages and focus on blind spots

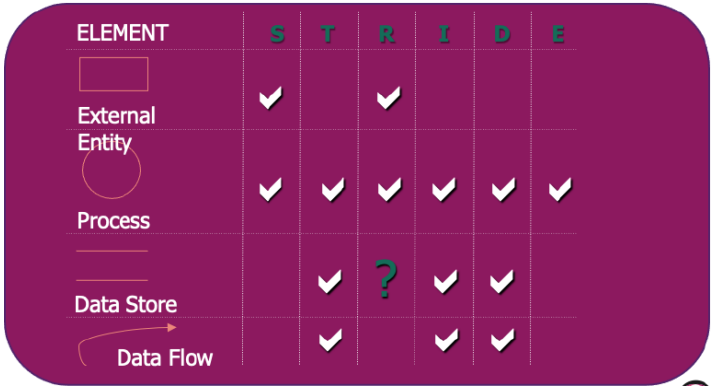
For Management

- Understand the risk involved in the application

4.4 STRIDE Cheatsheet

Threat	Property	Definition	Example
Spoofing	Authentication	Impersonating something or someone else.	Pretending to be any of billg, microsoft.com or ntdll.dll
Tampering	Integrity	Modifying data or code	Modifying a DLL on disk or DVD, or a packet as it traverses the LAN.
Repudiation	Non-repudiation	Claiming to have not performed an action.	"I didn't send that email," "I didn't modify that file," "I certainly didn't visit that web site, dear!"
Information Disclosure	Confidentiality	Exposing information to someone not authorized to see it	Allowing someone to read the Windows source code; publishing a list of customers to a web site.
Denial of Service	Availability	Deny or degrade service to users	Crashing Windows or a web site, sending a packet and absorbing seconds of CPU time, or routing packets into a black hole.
Elevation of Privilege	Authorization	Gain capabilities without proper authorization	Allowing a remote internet user to run commands is the classic example, but going from a limited user to admin is also EoP.

Spoofing	Authentication	To authenticate principals: • Cookie authentication • Kerberos authentication • PKI systems such as SSL/TLS and certificates To authenticate code or data: • Digital signatures
Tampering	Integrity	• Windows Vista Mandatory Integrity Controls • ACLs • Digital signatures
Repudiation	Non Repudiation	• Secure logging and auditing • Digital Signatures
Information Disclosure	Confidentiality	• Encryption • ACLS
Denial of Service	Availability	• ACLs • Filtering • Quotas
Elevation of Privilege	Authorization	• ACLs • Group or role membership • Privilege ownership • Input validation



4.5 Attack Trees in context

- Very useful for developing one threat where mitigation is not obvious
- Complement a threat model
- Can start a threat modeling
- Can be used in teams
- Often generate good discussions

5 Web Security

5.1 Introduction

- 56% of internet traffic is automated (hacking tools etc.)
- Critical systems on the web

5.2 OWASP

OWASP API Security Top 10	OWASP Top 10 (2017)
API1: Broken Object Level Authorization	A1: Injection
API2: Broken User Authentication	A2: Broken Authentication
API3: Excessive Data Exposure	A3: Sensitive Data Exposure
API4: Lack of Resources & Rate Limiting	A4: XML External Entities (XXE)
API5: Broken Function Level Authorization	A5: Broken Access Control
API6: Mass Assignment	A6: Security Misconfiguration
API7: Security Misconfiguration	A7: Cross-Site Scripting (XSS)
API8: Injection	A8: Insecure Deserialization
API9: Improper Assets Management	A9: Using Components with Known Vulnerabilities
API10: Insufficient Logging & Monitoring	A10: Insufficient Logging & Monitoring

5.3 Reasons for the many Security Incidents in the Web

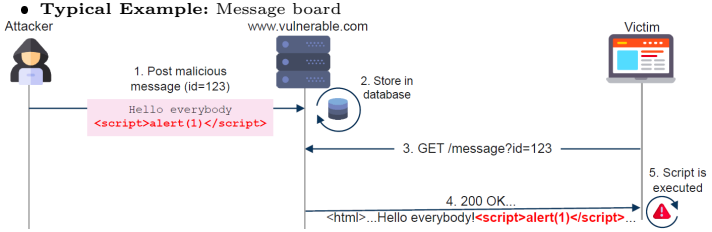
- The way how we develop software (human errors)
- Almost no regulation
- Time to market (no time to update/fix/test)
- Not aware of the risk
- It's difficult
- Attractive for hackers, easy to access targets on the web

5.4 XSS

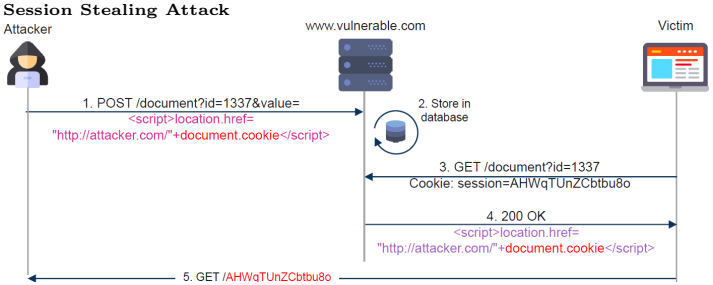
- Client-side code injection attack (victim's browser)
- Executing malicious scripts in the victim's browser
- Vulnerable applications deliver malicious scripts

5.4.1 Stored XSS

- Script is permanently stored on the target server
- Malicious script is executed for every visitor
- **Typical Example:** Message board

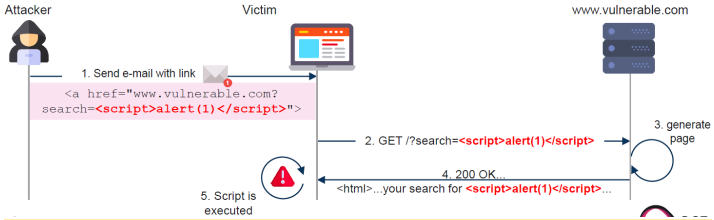


Session Stealing Attack



5.4.2 Reflected XSS

- Script is **not stored** on the target server
- Attacker usually needs to construct a malicious URL
- **Typical Example:** Search Form



5.4.3 DOM based XSS

- Vulnerability is the client side code rather than server-side code
- Attacker needs to construct an URL
- Parameter of the URL is not processed by the server, but executed by the browser directly

Vulnerable "Hello Joe" Page

```
<HTML><TITLE>Welcome!</TITLE>
Hello <script>
    var pos = document.URL.indexOf("name=") + 5;
    document.write(document.URL.substring(pos,document.URL.length));
</script>
</HTML>
```

Using the Page without "Hacking"

```
http://www.example.com/welcome.html?name=Joe
```

Exploiting the Page with DOM-based XSS string

```
http://www.example.com/welcome.html#name=<script>alert(document.cookie)</script>
```

5.4.4 Attack Vector Examples

- Script tags
- Event handler (*onLoad, onClick* etc.)
- Links
- InnerHTML assignment (DOM-based XSS)

5.4.5 Searching for XSS

- Play around with different strings in request params (script, ', ; etc.)
- In the page returned
  - Search for the presence of test strings
  - Check how the chars get filtered
  - Find the problem and test with new strings

5.4.6 The power of <script src >

- Bypassing the SOP

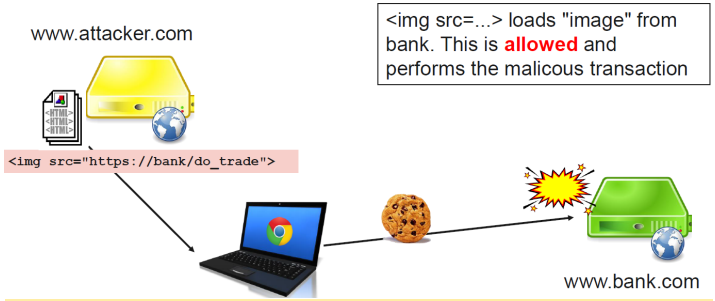
JavaScript from malware sites:
JavaScript from www.attacker.com IS **DENIED** from accessing resources from www.vulnerable.com when the JavaScript is loaded separately (e.g. separate browser tab)
Counter: JavaScript from www.attacker.com IS **ALLOWED** to access resources from www.vulnerable.com, if the script is loaded by www.vulnerable.com with <script src="..." >

5.4.7 XSS Protection

- Secure Programming
  - Input / Output encoding of user supplied data (HTML entities)
  - Input Validation (White/Blacklisting)
  - Using XSS Protection Features in Frameworks
  - Input sanitization of Web Frameworks
- HTTP Response Header
- Cookie HttpOnly
  - May prevent cookie / session stealing
  - Malicious JavaScript can still **ride the session** and exfiltrate and manipulate the DOM
- Browser XSS Filter (*X-XSS-Protection: 1; mode=block*)
  - Asks Browser to render a blank page if XSS is detected
  - Deprecated or never implemented for most browsers
  - Poor detection rate
- Content Security Policy (CSP)
  - Firewall for the Browser
  - Prevents code injection attacks like XSS
  - Supported by modern browsers
  - CSP defines, what is allowed to request from other domains
  - CSP comes as http response header or HTML tag
- Web Application Firewall
  - WAF Input Validation
  - Filtering HTTP Requests / Responses
  - Searching for XSS Payload

5.5 Cross-Site Request Forgery

- End User executes unwanted actions while authenticated



5.5.1 Pre-Requirement

- Know the target website / request
- Host the xsrf code somewhere
- Victim must have a valid session cookie

5.5.2 XSRF Protection

- Random Token
  - Form with a hidden field containing random token
- SameSite Cookie Attribute
  - Prevents browser from sending this cookie along with cross-site requests
  - Mitigates cross-origin information leakage
  - *lax / strict*
- CORS (only a mitigation)
  - Configure CORS on the API service
  - only allow requests from the fron-end origin
- Framework Support
  - AngularJS has built-in XSRF tokens on client side
  - Server side must be implemented by the dev

5.6 HTTP Security Headers

Strict Transport Security

- HSTS
- *Strict-Transport-Security: max-age=[ms]*
- Further requests in this time must occur over HTTPS
- Avoids all attacks based on HTTP downgrades

X-Content-Type-Options

- nosniff
- Problem: Wrong Content-Type in HTTP Response
- e.g. Blocks a request if the requested type is *style* and MIME type is not tt-ext/css"
- Supported by all major browsers

X-Frame-Options

- Prevent Website Framing
- Options: *deny / SAMEORIGIN*
- Affects *frame, iframe, embed, object*

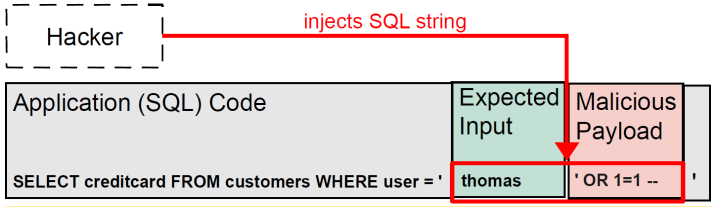
Referrer-Policy

- Control when URL information should be shared in the Referer header for other sites
- Options: *no-referrer / same-origin / orign-when-cross-origin*

Feature-Policy

- Allow and deny the use of browser features

5.7 SQL Injection



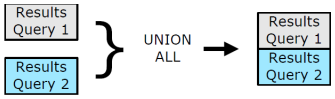
5.7.1 Bypass Authentication

- Login: meier
- Password: ' OR "='

```
SELECT username FROM Users
WHERE username='meier' AND password='' OR ''=''
```

5.7.2 Fetch Arbitrary Information

- Access to System and catalog tables
- Inject UNION statement
  - Assemble results from several SELECT queries
  - 2nd statement contains an arbitrary query




```
SELECT FirstName, LastName, Title
FROM Employees WHERE City = ' '
UNION ALL
SELECT OtherField, '', ''
FROM OtherTable WHERE ''=''
```

5.7.3 Hacking Database Example

1. What userID is running the MySQL database?
2. Disclose Password hash from database
3. Install Backdoor on Server through SQL Injection
4. Upload more hacker tools on the server

5.7.4 Mitigation

- Secure Programming
  - Prepared Statements / Stored Procedures
  - Escaping
- Error Handling
  - Do not disclose details
- Web Application Firewall
- DB least privileges

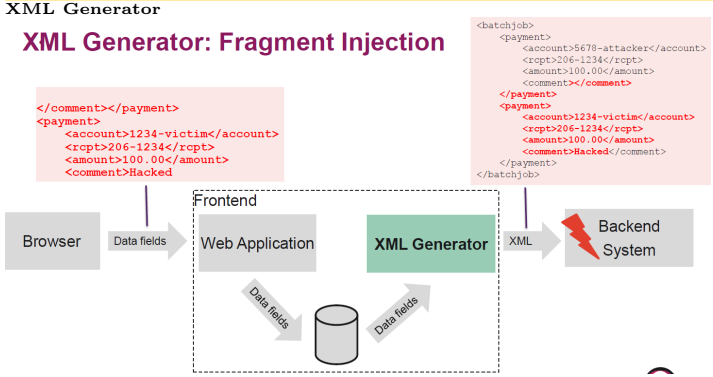
5.8 XML External Entity Attack

5.8.1 XML Security

- XML signatures
- XML encryption
- XML key management
- Security Assertion Markup Language
- XML access control markup language

5.8.2 Attack Vectors

XML Generator



Parser Attacks

- Attack Range
  - DoS
  - Inclusion of local files into XML documents (disclosure)
  - Port scanning from the system where the parser is located
  - Fetch data from local or remote systems

5.8.3 Mitigating XML Attacks

- Secure XML parser setup

6 Bugs

Security Bug:

- Vulnerability at the implementation level
- Easy to discover / remediate using mondern code review tools
- E.g. buffer overflows, race conditions, unsafe system calls

Security Flaw:

- Design level vulnerability
- Difficult / impossible to detect by automated tools
- Requires a manual risk analysis
- E.g. method overriding, error handling, type safety confusion

Security Defect = Bug + Flaw

- Bugs / Flaws which may lie dormant for years
- Can surface in fielded systems with major consequences

6.1 Example

```
1 read(fd, input0, sizeof(input0));
2 comparison = memcmp(input0, correctPasswd, strlen(input0));
3 if (comparison != 0)
4     return BAD_PASSWORD;
```

Source: Paul Kocher, Slashdot.org

- Line 1, Type Bug: Return Value is not assigned. This may be combined in an attack.
- Line 2, Type Bug: strlen() computes the length of the string input0 up to, but not including the terminating null character. So lets "hope" that read() in line 1 puts a terminating null char in...
- Line 2, Type Flaw: correctPasswd is stored in plain text
- Line 3, Typ Flaw: comparison validates if input0 is of length zero (empty input0), sic!

6.2 Memory Corruption

- Processes have their own address space others cant touch
- **Segmentation fault:** if a process tries to access memory outside its range

6.2.1 Buffer Overflows

char buffer[4]; buffer[4] = 'a'; // Undefined behaviour

- Can result in segmentation fault, if lucky
- Possible remote code execution, if unlucky

Solution:

- Check array boundary at runtime

6.2.2 Other memory corruption issues

```
1000 ...
1001 void f (){
1002   char* buf, buf1;
1003   buf = malloc(100);
1004   buf[0] = 'a';
     ...
2001   free(buf1);
2002   buf[0] = 'b';
     ...
3001   free(buf);
3002   buf[0] = 'c';
3003   buf1 = malloc(100);
3004   buf[0] = 'd'
3005
```

possible null dereference (if malloc failed)

potential use-after-free if buf & buf1 are aliased

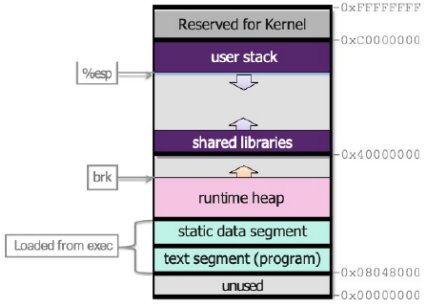
use-after-free; buf[0] points to de-allocated memory

use-after-free, but now buf[0] might point to memory that has now been re-allocated

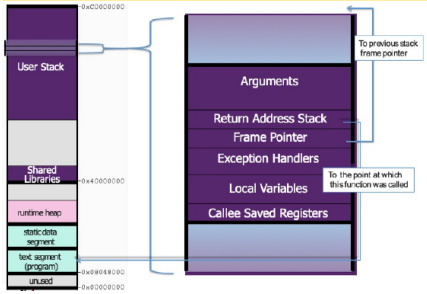
memory leak; pointer buf1 to this memory is lost & memory is never freed

6.3 Buffer Overflow

6.3.1 Process Memory Layout



6.3.2 Stack Frame



6.3.3 Problematic libc functions

- strcpy
- strcat
- gets
- scanf

Safe versions:

- strncpy
- strncat

6.3.4 Defenses

Compile:
Harden programs to resist attacks in new programs.

- Use a modern Programming Language
- Safe Coding Techniques
- Language Extensions / Safe Libraries
- Stack protection
  - Canaries: Check function entry and exit code to check for corruption
  - Random / Terminate canary

Run:
Aim to detect and abort attacks in existing programs.

- Executable Address Space Protection
  - Use virtual memory support to make some regions of memory non-executable
- ASLR: Address Space Layout Randomisation
- Non Executable Memory

6.4 Race Condition

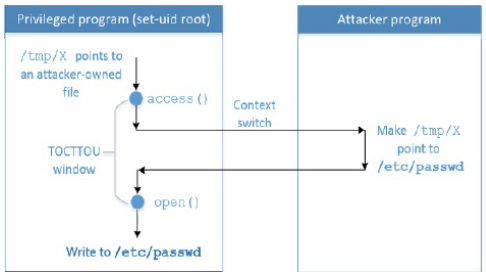
- Concurrency can lead to nondeterministic behaviour
- Software defects resulting from unanticipated execution ordering
- Necessary properties
  - Concurrency
  - Shared object
  - Change state

6.4.1 Race Window

- Occurs in a code segment that accesses the race object
- Window of opportunity for race condition
- Critical Section

6.4.2 Time of Check and Time of Use (ToCToU)

- Can occur during file I/O
- Forms a Race Window by first checking some race object and then using it



To win the race condition (ToCTToU window), we need two processes:

- Run vulnerable program in a loop
- Run the attack program

6.4.3 How to Exploit Race Condition

1. Choose a target file (e.g. */etc/passwd*)
2. Launch Attack
   - Attack Process
   - Vulnerable Process
3. Monitor the result (check timestamp)
4. Run the exploit

6.4.4 Mitigations

- Atomic Operations
- Repeating Check and Use
- Sticky Symlink Protection
- Principles of Least Privilege

listings graphicx

7 Integer Security

- Integer boundary concerns are often ignored
- Ariane 5 destroyed after integer overflow
- Results of integer overflow depend on the language and hardware

Integer Promotion Example: char c1, c2; c1 = c1 + c2;

7.1 Implicit Conversions

The sum of **c1** and **c2** exceeds the maximum size of **signed char**

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. c3 = -120;
5. cresult = c1 + c2 + c3;
```

However, **c1**, c2, and **c3** are each converted to integers and the overall expression is successfully evaluated.

The sum is truncated and stored in **cresult** without a loss of data

The value of **c1** is added to the value of **c2**.

7.2 Signed Integer Conversion

```
1. unsigned int l = ULONG_MAX;
2. char c = -1;
3. if (c == l) {
4.   printf("-1 = 4,294,967,295?\n");
5. }
```

The value of **c** is compared to the value of **l**.

Because of integer promotions, **c** is converted to an unsigned integer with a value of **0xFFFFFFFF** or 4,294,967,295

7.3 Overflow Examples

```
1. int i;
2. unsigned int j;
3. i = INT_MAX;  // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);
```

i=-2,147,483,648

```
6. j = UINT_MAX; // 4,294,967,295;
7. j++;
8. printf("j = %u\n", j);
```

j = 0

```
9. i = INT_MIN; // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);
```

i=2,147,483,647

```
12. j = 0;
13. j--;
14. printf("j = %u\n", j);
```

j = 4,294,967,295

7.4 Truncation Error Example

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. cresult = c1 + c2;
```

Adding **c1** and **c2** exceeds the max size of **signed char (+127)**

Truncation occurs when the value is assigned to a type that is too small to represent the resulting value

Integers smaller than **int** are promoted to **int** or **unsigned int** before being operated on

```
1.  bool func(char *name, long cbBuf) {
2.    unsigned short bufSize = cbBuf;
3.    char *buf = (char *)malloc(bufSize);
4.    if (buf) {
5.      memcpy(buf, name, cbBuf);
6.      if (buf) free(buf);
7.      return true;
8.    }
9.    return false;
10.  }
```

cbBuf is used to initialize **bufSize** which is used to allocate memory for **buf**

cbBuf is declared as a long and used as the size in the **memcpy()** operation

7.5 Sign Error Example

```
1. int i = -3;
2. unsigned short u;
3. u = i;
4. printf("u = %hu\n", u);
```

Implicit conversion to smaller unsigned integer

There are sufficient bits to represent the value so no truncation occurs. The two's complement representation is interpreted as a large signed value, however, so **u = 65533**

Sign Error Example (1)

```
1. #define BUFF_SIZE 10
2. int main(int argc, char* argv[]){
3.   int len;
4.   char buf[BUFF_SIZE];
5.   len = atoi(argv[1]);
6.   if (len < BUFF_SIZE){
7.     memcpy(buf, argv[2], len);
8.   }
9. }
```

Program accepts two arguments (the length of data to copy and the actual data)

len declared as a signed integer

argv[1] can be a negative value

A negative value bypasses the check

Value is interpreted as an unsigned value of type size_t

7.6 Integer Division

- Minimum integer value divided by -1
- $-2'147'483'648 / -1 = -2'147'483'648$

7.7 Negative Indices

```
1. int *table = NULL;\

2. int insert_in_table(int pos, int value){
3.   if (!table) {
4.     table = (int *)malloc(sizeof(int) * 100);
5.   }
6.   if (pos > 99) {
7.     return -1;
8.   }
9.   table[pos] = value;
10.  return 0;
11. }
```

Storage for the array is allocated on the heap

pos is not > 99

value is inserted into the array at the specified position

7.8 Mitigation

Strong Typing

- Provide better types
- Using unsigned type to guarantee positive values
- Does not prevent overflow

Abstract Data Type

- Create an abstract data type with private variables
- User can only changes values using public method calls
- Methods check valid range of input

Safe Int Class

- C++ template class
- Tests the values of operands before performing an operation

Testing

- Boundary checks

Source Code Audit

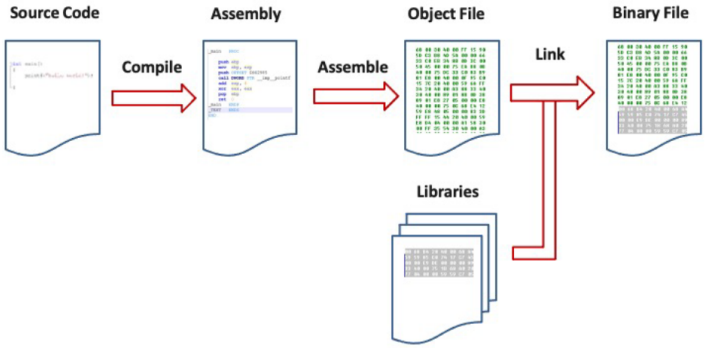
- Source code should be audited or inspected

8 Reverse Engineering

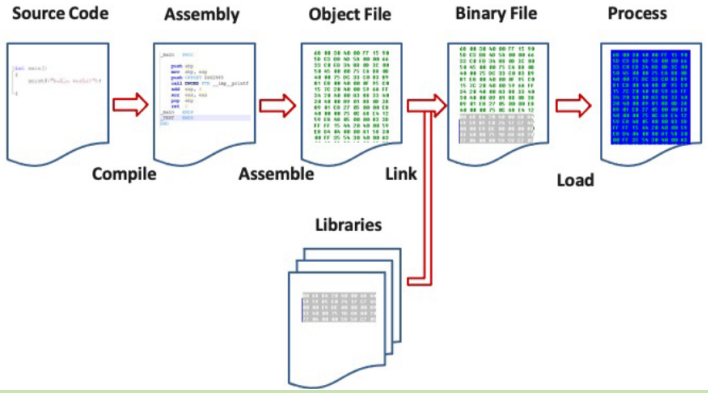
Software Reverse Engineering helps:

- understand malware
- understand legacy code
- remove usage restriction from software
- fond and exploit flaws
- cheat at games

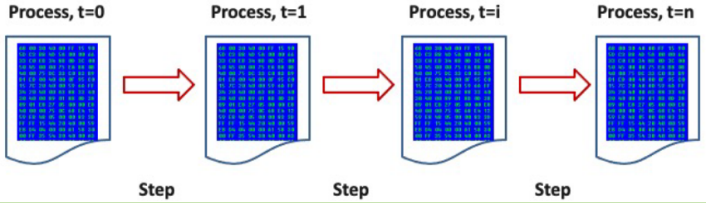
8.1 Compiling



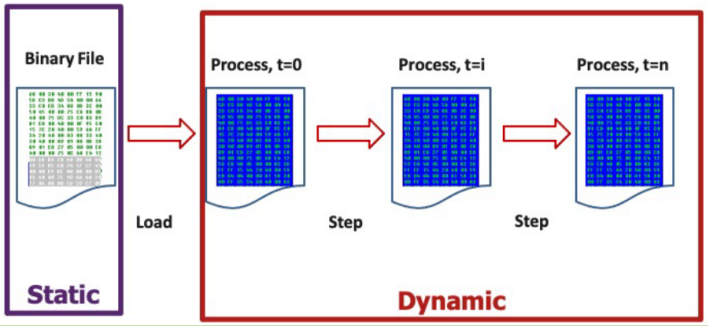
8.2 Loading



8.3 Running



8.4 Reverse Engineering Domain



8.5 Tools

Disassembler

- Converts exe to assembly (as best as it can)
- Cannot always disassemble 100% correctly
- Not possible to reassemble into working exe
- Gives static results
- Good overview of program logic
- Difficult to jump to specific place in the code

Debugger

- Step thru code to understand it
- Labor intensive
- Can set break points
- Can treat complex code as black box

Hex editor

- Patch (modify) exe file

Process Monitor, VMware, ...

8.6 Mitigations against Reverse Engineering

- Impossible to prevent
- Possible to make it more difficult
  1. Anti-disassembly: confuse static view of code
  2. Anti-debugging: confuse dynamic view of code
  3. Tamper resistance: code checks itself to detect tampering
  4. Code obfuscation: make code more difficult to understand

8.6.1 Anti Dissasembly

- Encrypted or packed object code
- False disassembly
- Self-modifying code
- Encryption prevents disassembly

8.6.2 Anti Debugging

- *IsDebuggerPresent()*
- Monitor for debug registers / breakpoints
- Debuggers do not handle threads well

Example:

- Programs gets *inst 1* and pre fetches following
- Debugger does not pre-fetch
- Overwrite later instruction after pre fetching
- Program without debugger will be okay
- Only works if segment of code is executed once

8.6.3 Tamper Resistance

- Make patching more difficult
- Code can hash parts of itself, hash checks later
- This approach is called *guards*

8.6.4 Code Obfuscation

- Make code hard to understand
- Opposite of good software engineering
- spaghetti code

int x,y; if((x-y)*(x-y) ¿ (x*x-2*x*y+y+y)) // always true **Code Bloating:**

1. Insert dead code
2. Adding zero effect operations
3. Transform the data (ASCII Chars instead of numbers)

9 Assurance

Confidence that an entity meets its security requirements based on evidence provided by applying assurance techniques.

Trusted System:

- Meets well defined requirements
- Evaluated by a credible body of experts
- Use specific methodologies to gather evidence

9.1 Issues with one-time evaluations

1. Requirements definitions may be flawed
2. System design can be flawed
3. Hardware implementation maybe faulty
4. Software implementation: well, errors, bugs
5. System use and operation errors
6. Willful system misuse
7. Hardware, communication or other equipment malfunction
8. Environmental problems
9. Evolution, maintenance, faulty upgrades and decomissions

9.2 Types of Assurance

Policy Assurance:
Evidence establishing security requirements in policy is complete, consistent, technically sound.
Design Assurance:
Evidence establishing design sufficient to meet requirements of security policy.
Implementation Assurance:
Evidence establishing implementation consistent with security requirements of security policy.
Operational Assurance:
Evidence establishing system sustains the security policy requirements during installation, configuration and day-to-day operation.

9.3 Assurance in Software Development LiveCycle (SDLC)

Conception

- Decision to pursue it
- Proof of concept to see if idea has merit
- High level requirements analysis
- Identify threats, assumptions

Manufacture

- Develop detailed plans for each group involved
- Implement the plans to create entity

Deployment

- Delivery Assure that correct masters are delivered to production
- Assure integrity of what is delivered to customers

Fielded Product Life

- Routine Maintenance
- Patching
- Support
- Decomissoning

9.4 Evaluation Models

9.4.1 Orange Book (TCSEC)

Collection of criteria used to grade or rate the security offered by a computer system product.
C1: Discretionary Protection

- Identification

- Authentication
- Discretionary access control

C2: Controlled Access Protection
- Object reuse and auditing

B1: Labled security protection
- Mandatory access control on limited set of objects
- Informal model of the security policy

B2: Structured Protections
- Trusted path for login
- Principle of least privilege
- Formal model of Security Policy
- Covert channel analysis
- Configuration management

B3: Security Domains
- Full reference validation mechanism
- Constraints on code development process
- Documentation, testing requirements

A: Verified Protection
- Formal methods for analysis, verification
- Trusted distribution

9.4.2 TCSEC Evaluation

Three Phases
- Design analysis
- Test analysis
- Final Review

Issues:
- Based heavily on confidentiality
- Not addressing integrity / availability
- Tied security and functionality together

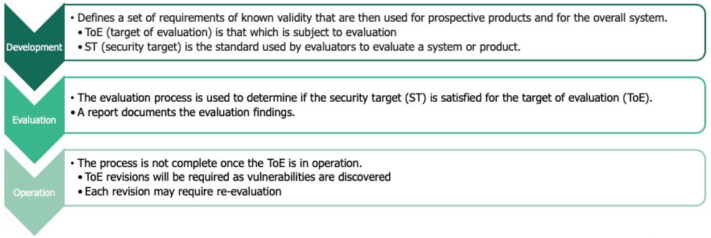
9.4.3 Common Criteria (CC)

- Provides IT security requirements for IT products
- Provides metric to quantify level of security
- Addressed Requirements:
  - Functional: define desired security behaviour
  - Assurance: indicating claimed security measures are effective and implemented correctly

Purpose:
- Provide consistent evaluation standards
- Improve the availability of evaluated systems
- Eliminate duplicating evaluations
- Improve the efficiency and cost-effectiveness

9.4.4 CC Process Approach

Target of Evaluation (ToE):
Name of the specific item that is the subject of the security evaluation.
Security Target (ST):
Standard used for the evaluation.



Assurance Levels (EAL:

- EAL 1: Functionally tested
- EAL 2: Structurally tested
- EAL 3: Methodically tested and Checked
- EAL 4: Methodically Designed, Tested and Reviewed
- EAL 5: Semiformal Designed, and Tested
- EAL 6: Semiformal Verified Design and Tested
- EAL 7: Formally Verified Design and Tested

9.4.5 CC Evaluation

1. Protection Profile
   - Implementation independant, domain specific set of security requirements
2. Security Target
   - Specific requirements used to evaluate system

9.5 Security Target

9.5.1 PP/ST Framework

- Product Approach usually for STs
- Define what product does
- Define existing documentation/assurance