1 Secure Software Principles

Three pillars
- Risk Management
- Touchpoints
- Knowledge

1.1 80% / 20% Problem Reduction

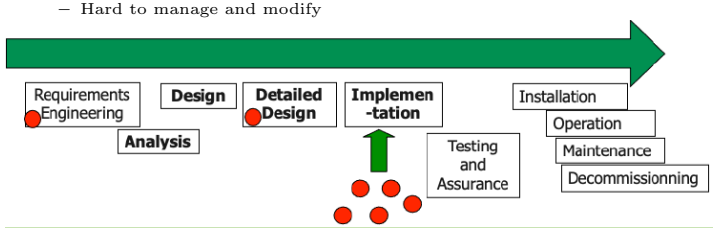
1. Secure the weakest link
2. Practice defence in depth
3. Fail securely
4. Follow the principle of least privilege
5. Compartmentalize
6. Keep it simple
7. Promote privacy
8. Remember that hiding secrets is hard
9. Be reluctant to trust (Off-the-shelf software)
10. Use your community resources (crypto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

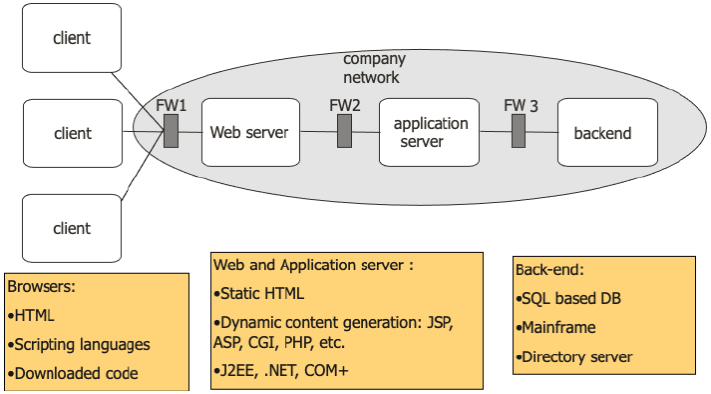
- For coarse-grained or simple requirements
  - Countermeasure can be realized in the operating system/middleware (SSL)
- For fine-grained and complex requirements
  - Implement as part of the application (hard to get right)

2.4 Other phases

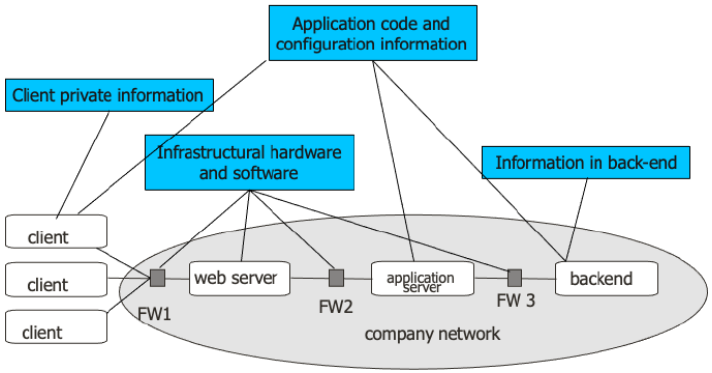
- Avoiding implementation vulnerabilities
- Security testing
- Automated patching

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

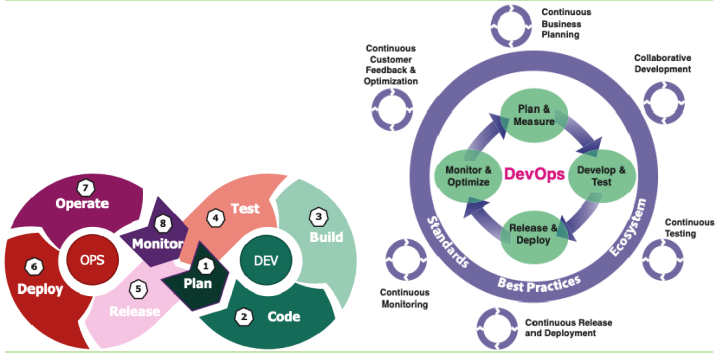
- Security technologies
- Quality improvements
  - Choice of programming language
  - Coding guidelines
  - Source code scanners
  - Security testing and audit
  - Code review

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



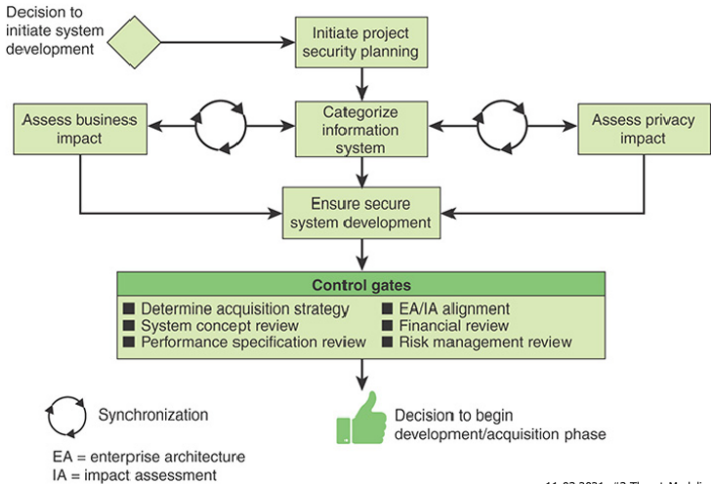
3.2 Four Major Activities

- 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

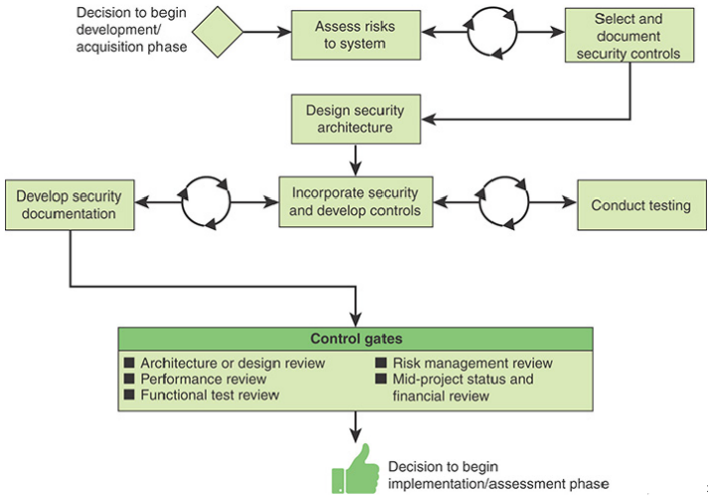
3.4.3 Ensuring secure system development

- 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

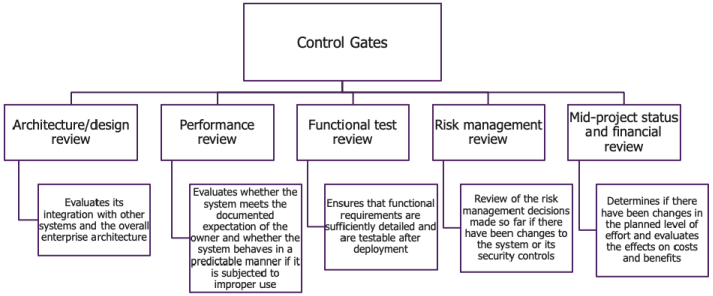
3.5 Develop & Test



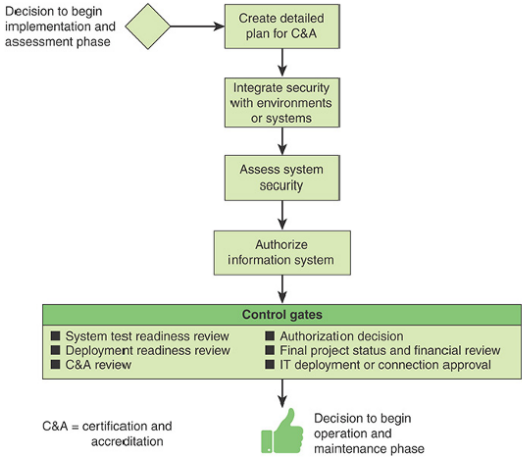
3.5.1 Designing the security architecture

- 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



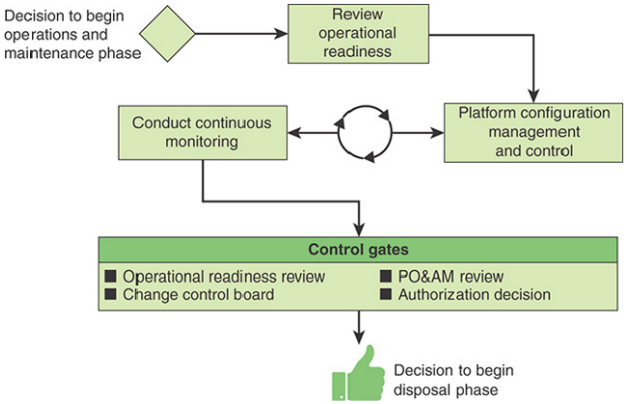
3.5.3 Begin Implementation / Assessment phase



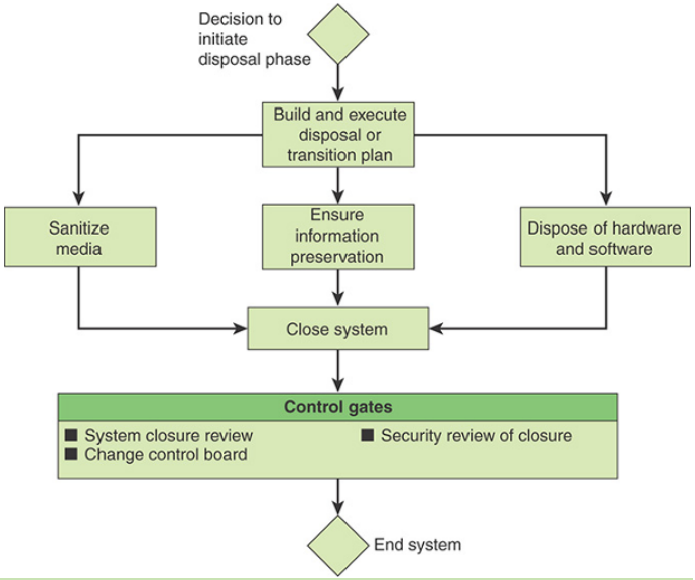
3.5.4 Control Gates

- 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



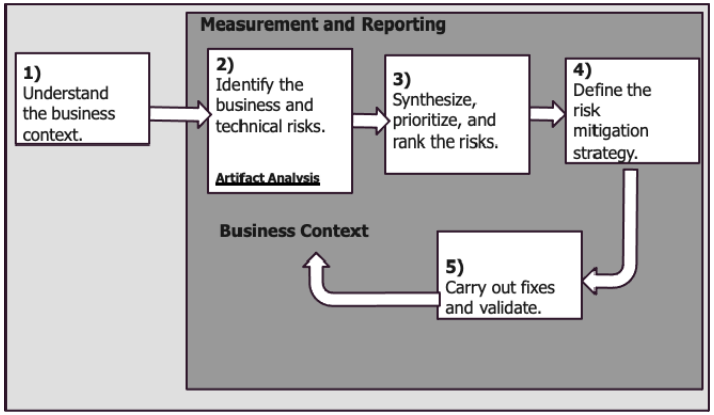
3.6 Continous Security

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

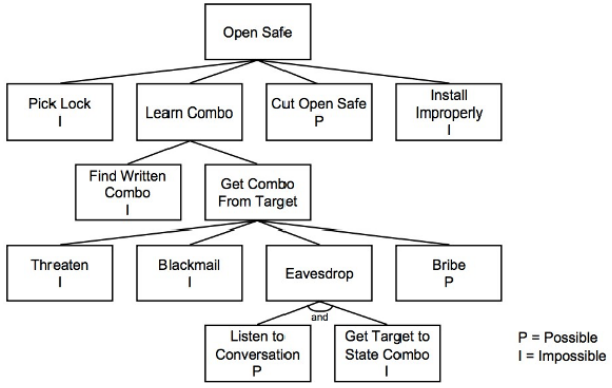
Construction:

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

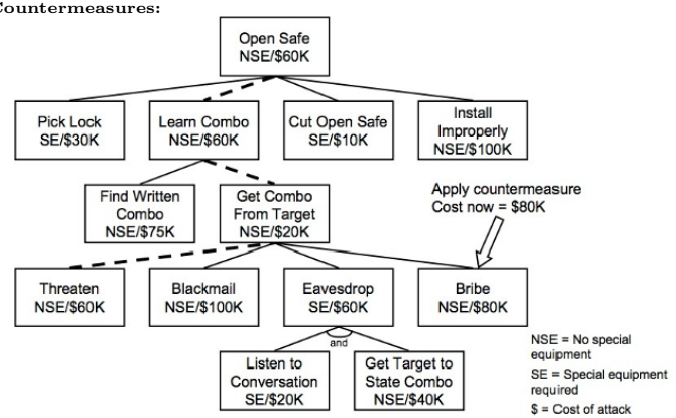
Usage:

- 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

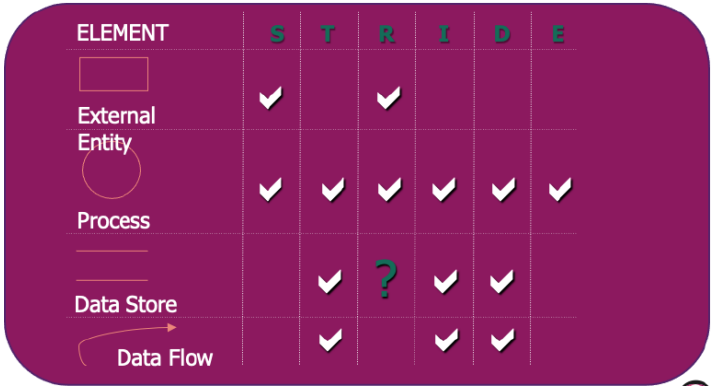
For Management

- 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

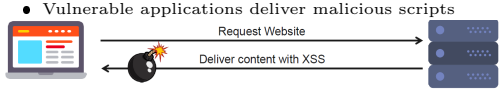
OWASP API Security Top 10	OWASP Top 10 (2017)
API1: Broken Object Level Authorization	A1: Injection
API2: Broken User Authentication	A2: Broken Authentication
API3: Excessive Data Exposure	A3: Sensitive Data Exposure
API4: Lack of Resources & Rate Limiting	A4: XML External Entities (XXE)
API5: Broken 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

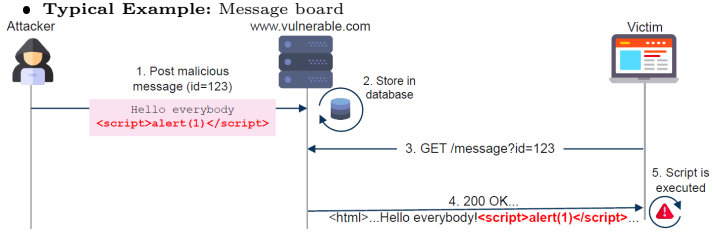
5.4 XSS

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

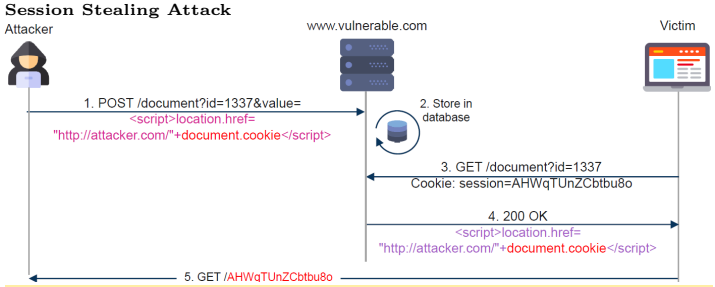


5.4.1 Stored XSS

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

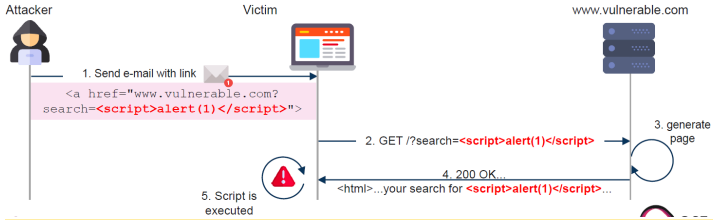


Session Stealing Attack



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

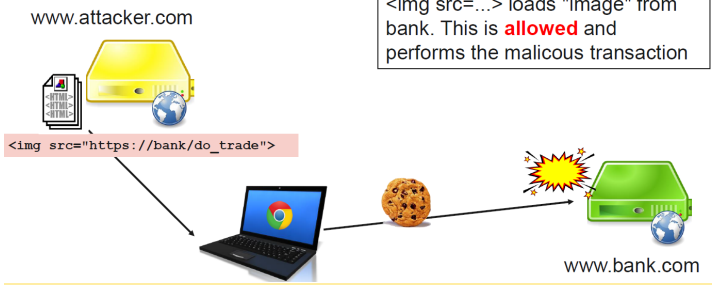
JavaScript from malware sites:
JavaScript from www.attacker.com IS **DENIED** from accessing resources from www.vulnerable.com when the JavaScript is loaded separately (e.g. separate browser tab)
Counter: JavaScript from www.attacker.com IS **ALLOWED** to access resources from www.vulnerable.com, if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

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

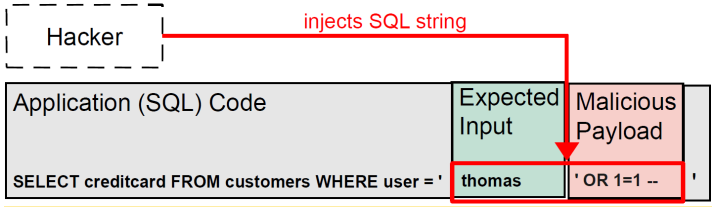
X-Content-Type-Options
- nosniff
- Problem: Wrong Content-Type in HTTP Response
- e.g. Blocks a request if the requested type is *style* and MIME type is not text/css"
- Supported by all major browsers

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



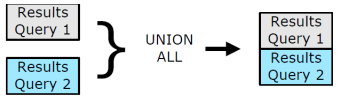
5.7.1 Bypass Authentication

- 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

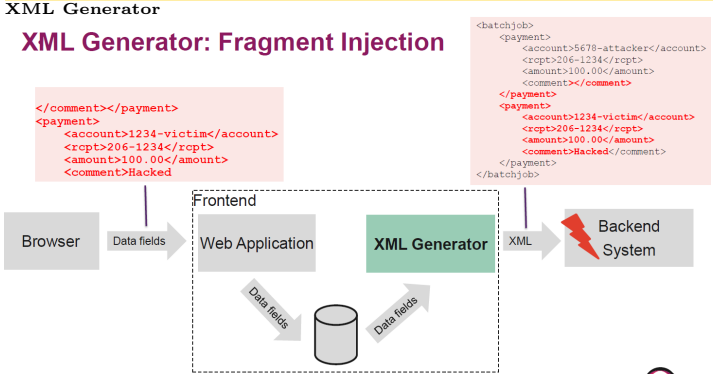
5.8 XML External Entity Attack

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:

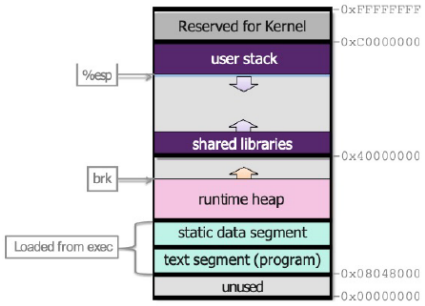
- Check array boundary at runtime

6.2.2 Other memory corruption issues

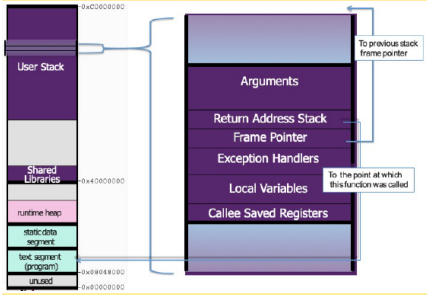
```
1000 …
1001 void f (){
1002   char* buf, buf1;
1003   buf = malloc(100);
1004   buf[0] = 'a';      <- possible null dereference (if malloc failed)
       ...
2001   free(buf1);
2002   buf[0] = 'b';      <- potential use-after-free if buf & buf1 are aliased
       ...
3001   free(buf);
3002   buf[0] = 'c';      <- use-after-free; buf[0] points to de-allocated memory
3003   buf1 = malloc(100); <- use-after-free, but now buf[0] might point to memory that has now been re-allocated
3004   buf[0] = 'd'       <- memory leak; pointer buf1 to this memory is lost & memory is never freed
3005
```

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

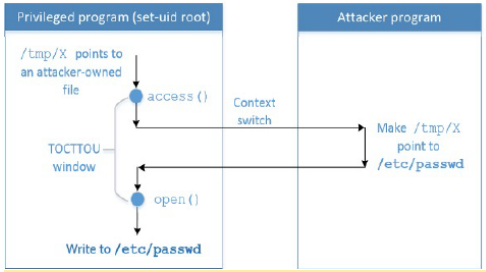
- Concurrency can lead to nondeterministic behaviour
- Software defects resulting from unanticipated execution ordering
- Necessary properties
  - Concurrency
  - Shared object
  - Change state

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);
6. j = UINT_MAX; // 4,294,967,295;
7. j++;
8. printf("j = %u\n", j);
9. i = INT_MIN; // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);
12. j = 0;
13. j--;
14. printf("j = %u\n", j);
```

i=-2,147,483,648

i=2,147,483,647

j = 0

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

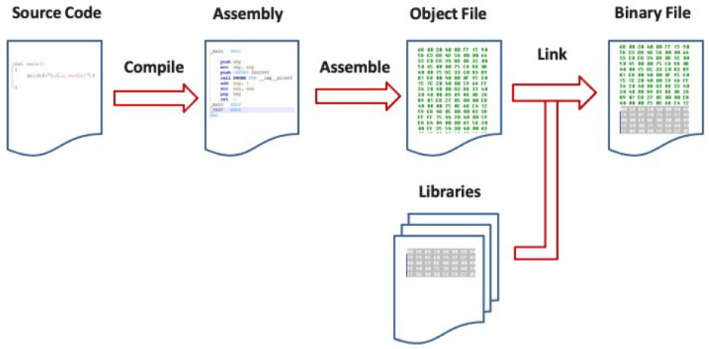
- Source code should be audited or inspected

8 Reverse Engineering

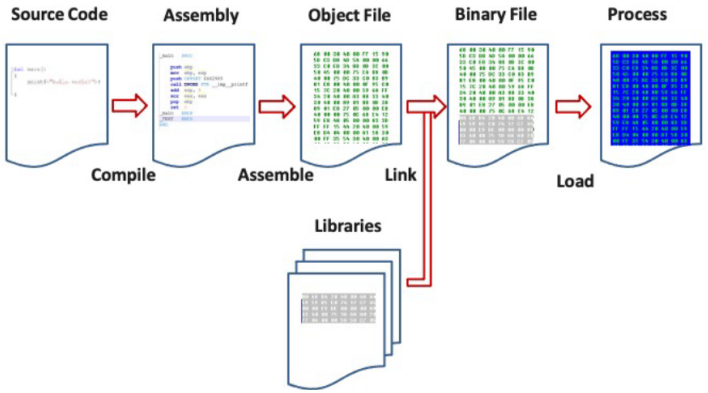
Software Reverse Engineering helps:

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

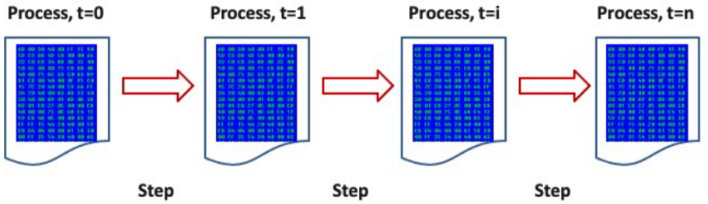
8.1 Compiling



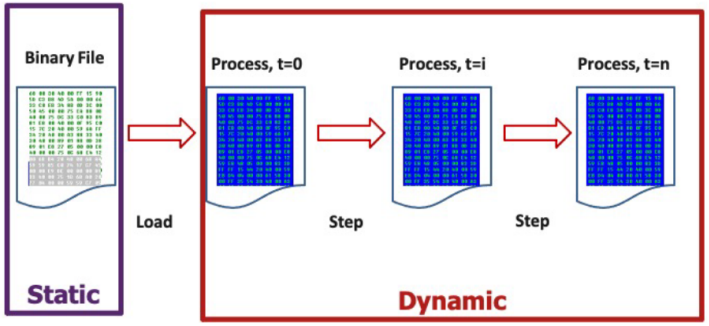
8.2 Loading



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)

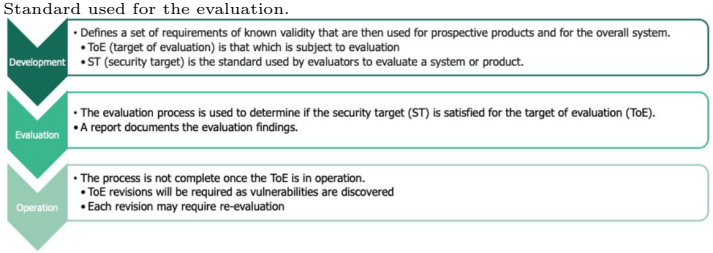
- Provides IT security requirements for IT products
- Provides metric to quantify level of security
- Addressed Requirements:
  - Functional: define desired security behaviour
  - Assurance: indicating claimed security measures are effective and implemented correctly

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

9.4.6 Example Threat in CC

Threat T1: A person not authorized to use the system gains access to the system and its facilities by impersonating an authorized user.

- Requirement IA1: A user is permitted to begin a user session only if the user presents a valid unique identifier to the system and if the claimed identity of the user is authenticated by the system by authenticating the supplied password.
- Requirement IA2: Before the first user/system interaction in a session, successful identification and authentication of the user take place.
- Assumption A1: The product must be configured such that only the approved group of users has physical access to the system.
- Assumption A2: Only authorized users may physically remove from the system the media on which authentication data is stored.
- Assumption A3: Users must not disclose their passwords to other individuals.
- Assumption A4: Passwords generated by the administrator shall be distributed in a secure manner.

Threat	Security Target Reference
T1	IA1, IA2, A1, A2, A3, A4

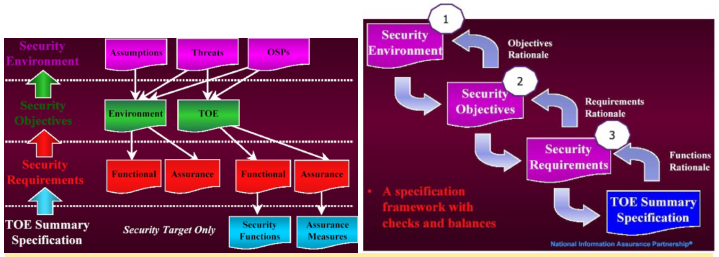
1. Referenced requirements and assumptions guard against unauthorized access.
   - Assumption A1 restricts physical access to the system to those authorized to use it.
   - Requirement IA1 requires all users to supply a valid identity and confirming password.
   - Requirement IA2 ensures that requirement IA1 cannot be bypassed.
2. Referenced assumptions prevent unauthorized users from gaining access by using valid user's identity and password
   - Assumption A3 ensures that users keep passwords secret
   - Assumption A4 prevents unauthorized users from intercepting new passwords when those passwords are distributed to users
   - Assumption A2 prevents unauthorized access to authentication information stored on removable media.

These justifications provide an informal basis for asserting that, if the assumptions hold and the requirements are met, the threat is adequately handled.

9.5 Security Target

9.5.1 PP/ST Framework

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



9.5.2 Security Environment

- **Secure usage Assumptions** aspects of the environment intended to use
- Describes the security aspects
- **Threats:** The ability to exploit a vulnerability by a threat agent
- **Organizational Security Policies:** Set of rules, procedures, practices imposed by an organization

9.5.3 Security Requirements

Functional Requirements

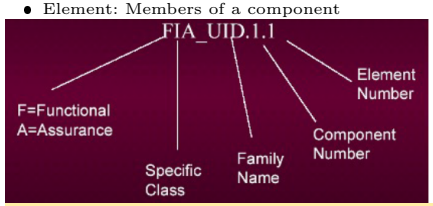
- Defining security behaviour
- Implemented requirements become security functions
- Examples:
  - Identification / Authentication
  - Audit
  - User Data protection

Assurance Requirements

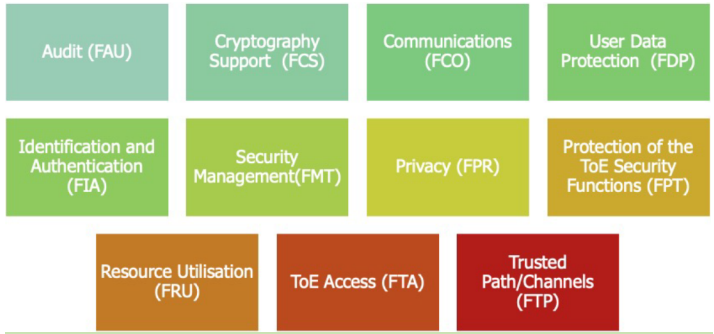
- Establishing confidence in security functions
  - Correctness of implementation
  - Effectiveness in satisfying security objectives
- Examples:
  - Development
  - Configuration Management
  - Life Cycle Support
  - Testing

9.5.4 Reading Guidance

- Class: Differ in coverage of security objectives
- Family: Differ in rigor or emphasis
- Component: Describes an actual set of security requirements
- Element: Members of a component



9.5.5 Security Functionality Classes



9.6 Design Document

- Provide basis for analysis
  - Informal
  - Semiformal
  - Formal
- Must include:
  - Security Functions
  - External Interfaces
  - Internal Design

9.6.1 Security Functions

- Identifies high-level security functions defined for the system
- Includes:
  - Description of individual functions
  - Overview of set of security functions, how they work together
  - Mapping of requirements, mapping functions to requirements

9.6.2 External Interfaces

High level description of external interfaces to system, component, subcomponent or module

1. **Component overview**: Identifying the component, its parent, how it fits into the design
2. **Data descriptions:** Identifying data types and structures needed to support the external interface
3. **Interface descriptions**

Example: Routine for error handling subsystem that adds an event to an existing log file

Interface Name
error_t add_logevent (handle_t handle, data_t event);
Input Parameters
handle — valid handle returned from previous call to *open_log*
event — buffer of event data with records in *logevent* format

Exceptions
Caller lacks permission to add to EVENT file
Inadequate memory to add to an EVENT file
Effects
Event is added to EVENT log.
Output Parameters
status status_ok — /* routine completed successfully */
no_memory — /* insufficient memory (failed) */
permission_denied — /* no permission (failed) */
Note
add_logevent is a user-visible interface

9.6.3 Internal Design

Describes internal structures and functions of components of system.

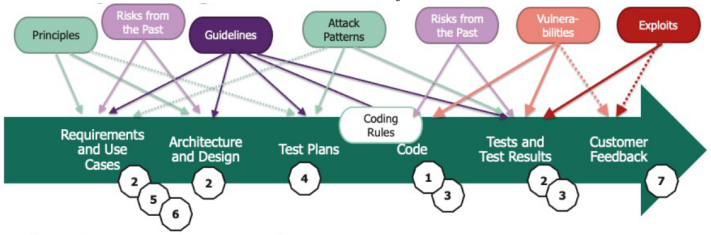
1. Overview of the parent component
2. Detailed description of the component
3. Security relevance of the component

9.7 Activities done to gain assurance

1. Analysis of the correspondence between TOE design representations
2. Analysis of the TOE design representations against the requirements
3. Analysis of functional tests coverage and results
4. Independent functional testing
5. Penetration testing
6. Verification of mathematical proofs
7. Analysis of guidance documents
8. Analysis of processes and procedures
9. Checking that processes and procedures are being applied

10 Security Testing

Process intended to reveal flaws in the security mechanisms.



1. Code Review Tools
2. Risk Analysis
3. Penetration Tests
4. Risk based Security Tests
5. Abuse (Test) Cases
6. Security Requirements (Tests)
7. Security Operation (Tests)

10.1 Verification in SDLC

Static analysis

- Approach for verifying software (including finding defects) **without executing software**
- Source code vulnerability scanning tools, code inspections, etc.

Dynamic analysis

- Approach for verifying software (including finding defects) **by executing software on specific inputs & checking results ("oracle")**
- Functional testing, web application scanners, fuzz testing, etc.

Hybrid analysis

- Approach combining static and dynamic approaches

10.2 Measurement Terminology

Analysis/tool report	Report correct	Report incorrect
Reported a defect	**True positive (TP):** Correctly reported a defect	**False positive (FP):** Incorrect report of a "defect" that is not a defect ("Type I error")
Did not report a defect	**True negative (TN):** Correctly did not report a given defect	**False negative (FN):** Incorrect because it failed to report a defect ("Type II error")

Issues:

- Too many false positives: Tool wasted my time
- Too many false negatives: Tool missed the important

10.3 Application to (Static) Application Security Testing Tools

Precision

- **Precision = #TP / (#TP+#FP)**
- Probability that a report is correct is called Precision
- High precision is a desired feature by developers (a low precision increased "Tool wasted my time"-perception)

Sensitivity

- **Sensitivity = #TP/(#TP+#FN)**
- Probability that a report will be generated if there is a defect is called Sensitivity
- Sensitivity is also called Recall, Soundness, Find Rate and True Positive Rate (TPR)

Harmonic Mean

- **Harmonic Mean = 2 x (Precision x Sensitivity) / (Precision + Sensitivity)**
- Reciprocal of average of reciprocals, a good averaging measure for many situations involving ratios; a common way to combine precision and sensitivity in one number, often also called F1 score

Discrimination Rate

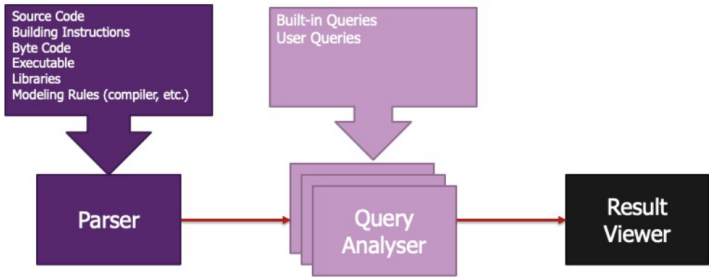
- Discrimination rate = #Discriminations / #Test_Pairs
- Given a pair of tests (one with defect, one without)
- Discrimination occurs if tool correctly reports flaw (TP) in test with flaw AND does not if there is no flaw (TN)

10.4 Static Analysis Tools

10.4.1 Static Application Security Testing (SAST)

- Based on white-hat / white-box
- Tester knows information about the system
- Examine source code at rest to detect and report security weaknesses
- Some tools run on source code, some on compiled code, some on both

Functional Components of a SAST Tool



Data Flow Analysis Techniques compiled into SAST

1. Control Flow Graph: Follow the control flow to identify dangerous sequences
2. Tainting Analysis: Identify variable that have been tainted with user input, follow them if passed without sanitization
3. Lexical Analysis: Tokenize source code to make it easier to analyse

10.5 Dynamic Analysis Tools

10.5.1 Web Application Scanners

- Pretend Browser
- Goes through the various web forms and links
- Sends in attack-like and random data, tries to detect problems
- Easy and quick to use
- OWASP ZAP

10.5.2 Fuzzing

- Provide a large number of random inputs
- Monitor program results and not if the final answer is correct
  - Crashes
  - Fails code assertions
  - Memory leaks

10.5.3 Test Data Creation Techniques

1. Black Box (fully random): Feed the program random inputs. Very easy but rather inefficient
2. Mutation based: Mutate input samples to create test data
3. Generation based: Create test data based on model of input

Imporvements:

- Constraints: Generating tests that execute previously unused code paths
- Heuristics: Create likely security vulnerability patterns
- Variate on the type of input data

10.6 Code Coverage

- **Statement coverage:** Which percentage of statements have been executed
- **Branch coverage:** Which percentage of branch options have been executed

10.7 Software Composition Analysis (SCA)

- Historically developed to check legal status of embedded software libraries
- Today: powerful tool to get inside about the propagation of vulnerabilities in 3rd party libs

10.8 Scanning for Secrets

- Analysis technique to look for unintended revelation of secrets

graphicx

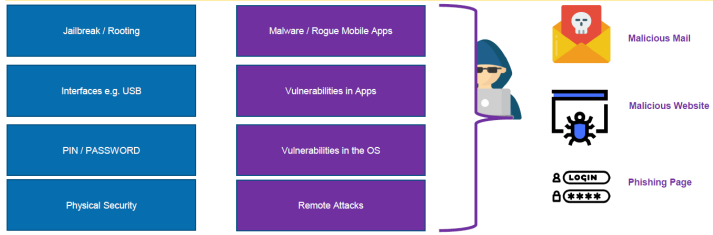
11 Mobile Security

11.1 Introduction

Challenges

- Physical security
  - Mobile phones can be lost or stolen
  - Sensitive data on device
  - Bluetooth, NFC
- BYOD - Bring your own device
  - Policies
  - Mobile Device Management
- Updates
  - Delayed or not available for older / unsupported devices

11.1.1 How to Attack a Smartphone



11.1.2 Mobile Security Threats

- Vulnerabilities in the Mobile Operating System / Bundled Apps
- Vulnerabilities in Apps
- Untrustworthy devices or software

11.1.3 What malicious actors do

Unique to Smartphones

- Premium SMS messages
- Identify location
- Record phone calls
- Record/Forward/Intercept SMS (2FA)
- Encrypt Shared Data = Ransomware

Similar to Desktop

- Connects to botmasters (DDoS, Crypto-Mining)
- Steal data
- Phishing
- Intercept communication or UI

11.2 Overlay Attacks using Process Scanning

- Malware determines the top most running app
- Display a fake screen that is similar to the original app
- Used for Automated Transfer Attacks via banking app

11.3 Secure Programming Mobile Apps

Use maximal Security against:

- Lost smartphone
- Weak device config (no PIN)
- Rouge mobile Apps
- Network Man-in-the-Middle
- Client Side Code Injection
- Insecure Data Storage
- App is running on a hacked or untrusted device

11.4 iOS

- Derived from macOS
- Unix-like (based on Darwin)

Common Programming Languages

- Swift
- Objective C
- C

Restricted App Distribution

- App Store by Apple
- Reviewed by Apple
- Code Signing

Security

- Sandbox Mechanism
- File Encryption / Data Protection API
- KeyChain

11.4.1 Apple Plattform Security

- Online Documentation
- Security by design, integrated into the core
- Combination of hardware and software security
  - **Secure Enclave (Processor)**
  - dedicated co-processor
  - used for TouchID and KeyChain

11.4.2 iOS Operating System Security

Secure Boot Chain / Root of Trust / Code Signatures

- Prevent execution of third-party / untrusted software

System Software Authorization

- Prevent downgrade of operating system and firmware

Secure Enclave / Secure Element

- Isolate crypto operations (Touch / FaceID)

Lockdown Protocol

- Secures the pairing / communicating with a PC

File System Encryption / Effaceable Storage

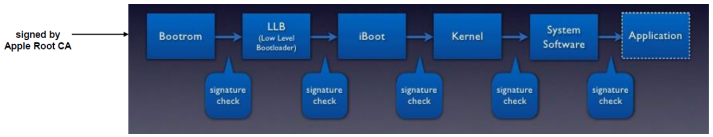
- Prevent unauthorized access to stored data
- Secure data deletion

Sandbox Mechanism / File Data Protection / Keychain

- Application Isolation / Least Privileges

11.4.3 Secure Boot Chain

- Startup in steps
- Each step checks integrity of following step: **Chain of Trust**
- Bootrom exploits cannot be patched



11.4.4 App Sandbox

- Each app has its own sandbox
- Dedicated directory with subfolders
- Isolates data and code
- Other apps have no access
- Sharing data must be implemented explicitly
- Limits damage in case app is compromised

11.4.5 Permission Model

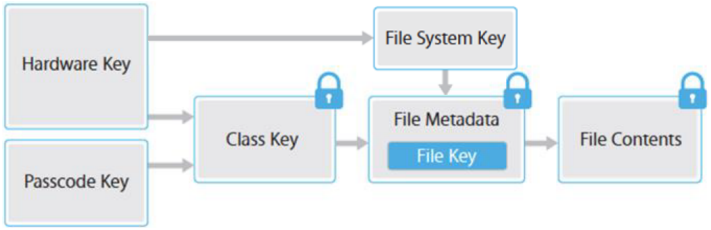
- Permissions are requested when used
- Not at installation time
- User can deny individually
- Can be changed anytime in settings
- Usage description since iOS 10

11.4.6 Data Protection API

- **Every** file is encrypted using AES
- Hierarchy of keys
- Key handling and en/decryption is part of the secure enclave
- Keys are never exposed to the operating system or apps

Keys:

- Hardware Key: embedded into Crypto component
- Passcode Key: derived from Passcode / PIN / Fingerprint
- Class Key: encrypted with the above
- File Key: encrypted with class key **unique for each file**
- File System Key: protected using hardware key



Data Protection Classes

- Different Class Keys
  - Always
  - After first unlock (default as of iOS 7)
  - While locked
  - When unlocked

Availability	File Data Protection
When unlocked	NSFileProtectionComplete
While locked	NSFileProtectionCompleteUnlessOpen
After first unlock	NSFileProtectionCompleteUntilFirstUserAuthentication
Always	NSFileProtectionNone

11.4.7 Keychain API

- SQLite database
- Used to store secrets
- Keychain respects the sandbox
- Protected by hardware

Keychain Availability Classes

- Similar classes as for Data Protection API
  - When Passcode Set
  - When Unlocked
  - After First Unlock
  - Always
- Supports device binding using the ThisDeviceOnly suffix

Availability	Keychain Data Protection
When unlocked	kSecAttrAccessibleWhenUnlocked
While locked	N/A
After first unlock	kSecAttrAccessibleAfterFirstUnlock
Always	kSecAttrAccessibleAlways
Passcode enabled	kSecAttrAccessible-WhenPasscodeSetThisDeviceOnly

Keychain Access Control

- Access Control flags define how the item can be accessed
- kSecAccessControl, can combine options using and / or

11.4.8 TouchID

- Stored in CPU (Secure enclave)
- Not stored server side

11.5 Android

Common Programming Languages

- Kotlin
- Java
- C

Restricted App Distribution

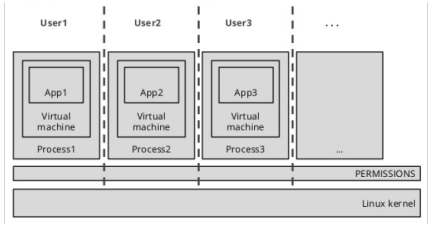
- Play store by Google
- Alternative stores possible
- **no** review
- Coding signing

Security

- Sandbox Mechanism
- File Encryption / Data Protection API
- Keychain

11.5.1 App Sandbox

- Each app has own sandbox
- Separation using traditional linux permissions
- Each app has its own JVM
- Isolates data and code - a bit simpler than iOS
- Limits damage in case app is compromised



11.5.2 Permission Model

Android 5

- Permissions are requested when installed
- take it or leave it
- all-or-nothign

Since Android 6

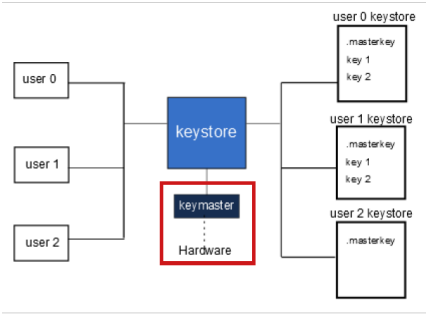
- Approval at runtime
- Can be changed in preferences

11.5.3 Data Encryption

- Full Disk Encryption (FDE)
- File Based Encryption (FBE)

11.5.4 Keystore API

- Container for credentials
- Can be hardware-backed



11.5.5 Fingerprint API

- Combination with KeyStore
- UI in full control of app

11.5.6 Google Play Protect

- Malware scanning
- Lost phone tracking / blocking
- **Constantly** scanning apps

11.6 Cross-Plattform & Hybrid Apps

11.6.1 Cross-Plattform Apps

Often used when trying to reduce software development time by:

- Sharing code between multiple platforms
- Using established programming languages
- Examples:
  - Xamarin
  - React Native
  - Flutter

11.6.2 Hybrid Apps

- Mobile App development using HTML, JS, CSS
- System component renders content
  - iOS: WKWebView
  - Android: WebView
- Interactions with native code

Pros

- Lightweight applications
- Portable (write once, use on different plattforms)
- Reuse code from web world

Cons

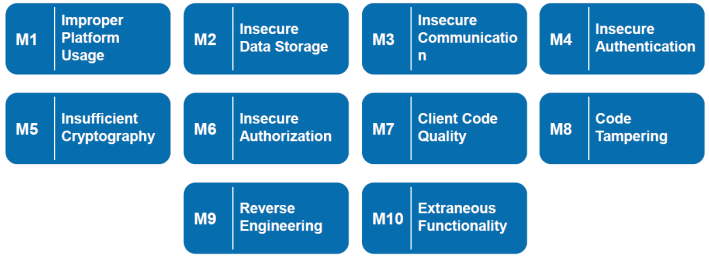
- Often the UI does not fit the platform's standard
- Performance
- Security - Same risks as for Web Apps

11.6.3 WebView Hardening

- Proper SSL handling and checking
- Implement a white-list for allowed URLS (https only)
- Block other protocols
- Obfuscate (Javascript) code
- Restrict JavaScript and/or file access if possible

graphicx

11.7 OWASP Mobile Top 10



11.7.1 Improper Platform Usage

Risk

- Violation of published guidelines
- Violation of convention or common practice
- Unintentional Misuse

Examples

- Grant app too many permissions
- App local Storage Instead of Keychain

Countermeasures

- Secure coding
  - Guidelines by the manufacturer
  - Best-practices
- Secure configuration
  - Review
  - Least privilege

11.7.2 Insecure Data Storage

Risk

- Insecurely storing data
  - Databases
  - Log files
  - Cookies
  - SD card
  - Cloud
- Data leakage
  - in OS, frameworks, hardware
  - without developers knowledge

Channels

- Logs
- Screenshots
- Keyboard (autocomplete function)
- Clipboard
- Web Data (Cache, Cookies, Local Storage)
- Inter Process Communication

Countermeasures:

- Critical data
  - No logging, no caching
  - If necessary, mask data
- Logs: disable debug logs in production
- Keyboard: disable autocomplete
- Clipboard: disable copy/paste
- Screenshots: use FLAG_SECURE property

11.7.3 Insecure Communication

Risk

- Cleartext communication of sensitive data
- Eavesdropping
- TLS/SSL issues
- Weak negotiation or ciphers
- On any type of communication

Countermeasures:

- Use SSL/TLS whenever possible
- Use good certificates from trusted issuer
- Don't send sensitive data over unsecure channels
- Certificate Pinning
  - Extended validation of SSL server certificate
  - Store server certificate explicitly in app
  - Compare on each connection
  - Certificate renewal must be handled

11.7.4 Insecure Authentication

Risk

- Problems with authentication
  - Missing authentication
  - Weak authentication
  - Spoofing
  - Stored passwords
- Problems with session management
  - Predictable Session ID
  - No Session Timeout

Countermeasures

- Authentication
  - Perform authentication (also) on server-side
  - Enforcing strong passwords
  - Use fingerprint API
- Session handling
  - Implement session timeout on server side
  - If data must be stored on client side: encrypted

11.7.5 Insufficient Cryptography

Risk

- Weak ciphers / algorithms
- Self made cryptography
- Weak keys
  - Insufficient length
  - Stored in an unsecure place
  - Hardcoded keys

Countermeasures

- Avoid storing sensitive data on mobile device
- Choose proven algorithms / key sizes
- Never ever implement crypto yourself
- Use hardware-supported crypto

11.7.6 Insecure Authorization

Risk

- Grant access to unauthorized users
- Grant operations a user is not entitled for
- Examples:
  - Insecure Direct Object Reference
  - Hidden Endpoints (Access to backend API not protected)

Countermeasures

- Verify roles/permissions on server side
- Proper Access control on server side

11.7.7 Client Code Quality

Risk

- Buffer overflows
- Format string vulnerabilities
- Injection vulnerabilities
- XSS
- Code executing on the client side

Countermeasures

- SQL injection: validation, prepared statements
- XSS: validation, output encoding
- Coding patterns, code analysis tools
- Validate length of buffers
- Compiler Settings

11.7.8 Code Tampering

Risk

- Counterfeit applications
  - re-packaging of original app
  - malicious code injected
- Insecure installation of apps (third party stores)
- Techniques:
  - Binary patching / monkey patching
  - Resource patching
  - Method hooking / swizzling
  - Jailbreaking / Rooting

Countermeasures

- Detect code modifications
- React to code integrity violations
- Jailbreak detection

11.7.9 Reverse Engineering

Risk

- Theft of intellectual property
- Disclosure of critical parts

Countermeasures

- Code Obfuscation
- Anti-Debug mechanisms
- Use C code

11.7.10 Extraneous Functionality

Risk

- Functionality enabled that was not intended to be released
  - Hidden Backdoors
  - Dev switches
  - Disabled authentication
- Sensitive information
  - in comments
  - in debug logs

Countermeasures

- Check config files for hidden switches
- Make sure dev code is not included into production
- Don't put sensitive data into comments
- No debug logs in production
- Manual secure code review

11.8 Secure Coding Checklist

Data at Rest

- DataProtection API & Keychain API
- Backup Exclusion
- Caching

Data in Motion

- Transport Security
- Certificate Pinning

Data Leakage

- Prevent Side-channel leakage (logs, keyboard, etc.)

Input Validation

- Input sanitization
- Format strings

Code Protections

- Code Obfuscation
- Stack protection
- Anti-Debug controls
- Compiler settings
- Code Injection Checks

Web-View hardening

- Disable local file access
- Disable plugins
- Disable JavaScript
- URL whitelisting

Environmental Integrity

- Jailbreak / Rooting detection
- Version control / mandatory updates

12 Summary

12.1 Guiding Principles

1. Secure the weakest link
2. Practice defense in depth
3. Fail securely
4. Follow the principles of least privilege
5. Compartmentalize
6. Keep it simple
7. Promote privacy
8. Remember that hiding secrets is hard
9. Be reluctant to trust
10. Use your community resources

12.2 Secure SDLC

How to address security in the software engineering process

- Enriching it with activities for security
- Embracing state-of-the-art techniques

12.3 Threat Modeling

DevOps: Continous Security NOT Security on top or after the fact.

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

12.3.1 WHY?

For Developers

- If used throughout the development life cycle, exposure to security risks can be minimized
- Identifies and prioritize the threats found, and categorize the threats, making issues that may happen more 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 to be released application