Security Engineering fiche

Pierre Colson

December 2022

Contents

Markdown version on *github*
Compiled using *pandoc* and `gpdf` *script*

Introduction

- Security is usually added on, not engineered in
  - Standard security properties (CIA) concern *absence of abuse*
    - **Confidentiality**: No proper disclosure of information
    - **Integrity** No proper modification of information
    - **Availability** No proper impairment of functionality/service
- Sofware is not *continuous*
- Hackers are not typical users
  - A system is **safe** (or **Secure**) if the environment cannot cause ot to enter an unsafe (insecure state)
    - So, abstractly, security is a *reachability* problem
- The adversary can exploit not only the system but also the world
- Security Engineering = Software Engineering + Information Security
- **Software Engineering** is the application of systematic, quantifiable approaches to the development, operation, and maintenance of software; i.e applying engineering to software
- **Information Security** focuses on methods and technologies to reduce risks to information assets
- **Waterfall model**

- *Requirement engineering*: What the system do ?
- *Design*: How to do it (abstract) ?
- *Implementation*: How to do it (concrete) ?
- *Validation and verification*: Did we get it right ?
- *Operation and maintenance*
- Problems
  * The assumption are too strong
  * Proof of concept only at the end
  * Too much documentation
  * Testing comes in too late in the process
  * Unidirectional
- **Summary**
  - Methods and tools are needed to master the complexity of software production
  - Security needs particular attention
    * Security aspects are typically poorly engineered
    * Systems usually operate in highly malicious environment
  - One needs a structured development process with specific support for security

Requirements Engineering

- **Requirements engineering** is about elicting, understanding, and specifying what the system should do and which properties it should satisfy
- **Requirements** specify how the *system should and should not behave* in its intended environment
  - **Functional requirements** describe *what system should do*
  - **Non-functional requirements** describe *constraints*
- Security almost always conflicts with *usability* and *cost*
- Analysis → Specification → Validation → Elicitation → Analysis . . .
  - **Elicitation**: Determine requirements with stakeholders
  - **Analysis**: Are requirements clear, consistent, complete
  - **Specification**: Document desired system behavior
    * *Functionality*: what the softwate should do
    * *External interfaces*: how it interacts with people, the system's hardware, other software and hardware
    * *Performance*: its speed, availability, response time, recovery tim eof various software functions, etc
    * *Attributes*: probability, correctness, maintaintability, security, etc.
    * *Design constraints imposed on the implementation*: implementation language, resource limit, operating system environment, any required standard in effect, etc.
  - **Validation**: Are we building the right system?
- Standards and guidlines provide good strating points, but they must be refined and augmented to cover concrete systems and the infornations they process
- **Authorization policy**: knowing which data is critical is not wnough
  - Information access policy (Confidential, Integrity)
  - Good default is base on *least-priviledge*
- **Summary**
  - Security requirements are both functinal and non-functional
  - Standards and guidlines help with the high level formalization
  - Models help to concretize the details
    * However full details usually only present later after design
  - Models also useful for risk analysis

Modeling

- Overall goal: specify requirements as precisely as possible
- A **model** is a construction or mathematical object that describes a system or its properties
- The construction of models is the main focus of the *design* phase
- **Entity/Relationship modeling (E/R)**
  - Very simple language for data modeling
    - Specify set of (similar) data and their relationships
    - Relations are typically stored as tables in a data-base
    - Useful as many systems are data-centric
  - Three kinds of objects are visually specified
    - **Entities**: sets of individual objects
    - **Attributes**: a common property of all objects in an entity set
    - **Relations**: relationships between entities
  - *Pros*
    - 3 concepts and pictures $/Rightarrow$ easy to understand
    - Tool supported and successful in practice, E/R diagrams mapped to relational database schemes
  - *Cons*
    - Not standardized
    - Weak semantics: only defines database schemes
    - Say nothing about how data can be modified
- **Data-flow diagrams**
  - Graphical specification language for functions and data-flow
  - Useful for requirements plan and system definition
  - Provides a high level system description that can be refined later
- **Unified Modeling Language (UML)**
  - 14 languages for modeling different views of systems
  - **Static models** describe system part and their relationships
  - **Dynamic models** describe the system's (temporal) behavior
- **Use Cases** key concepts
  - *System*: the system under construction
  - *Actor*: users (roles) and other systems that may interact with the system
  - *Use case*: specifies a required system behavior according to actors' need (*textually*, *activity diagram*)
  - Relations between actors: *Generalization/specialization*
  - Relations beween use cases:
    - *Generalization/specialization*
    - *Extend* (one use case extend the functionality of another)
    - *Include*
- **Activity diagrams**
  - *Action*: a single step, not further decomposed
  - *Activity*:
    - Encapsulates a flow of activities and actions
    - May be hierarchically structured
  - *Control flow*: edges ordering activities
  - *Decision*: a control node chossing between outgoing flows based on guards
  - *Object flow*: an adge that has objects or data passing along it
- **Class Diagram**
  - *Class*: describes a set of objects that share the same specifications of features, constraints, and semantics
  - *Attributes*:
    - A structural feature of a class
    - Define the state (date value) of the object
  - *Operation* (or *methods*):
    - A behavior feature of a class that specify the name, type, parameters and any constraints for

    invocation
    * Define how objects affect each other
  - *Association*:
    * Specifies a semantic relationship between typed instances
    * Relates objects and other instances of a system
    * They can have properties
  - *Generalization*:
    * Relates a specific classifier to a more general classifier
    * Relation between a general thing (*superclass*) and a specific thing (*subclass*)
  - A *class diagram* describes the **kind of objects** in a system and their different **static relationships**
  - Kind of relationships include:
    * *Association* between objects of a class
    * *Inheritance* between classes themsleves
- **Component Diagram**
  - *Component*:
    * Modular part of a system that encapsulates its contents and whose manifestation is repleable within its environment
    * Behavior typically implemented by one or more classes of sub-component
  - *Provided interfaces*: interfaces implemented and exposed by a component
  - *Required interfaces*: interfaces required to implement component's behavior
  - *An assembly connector*: links an interface provided by one component to an interface required by another component
  - *Ports*: named sets of provided and required interfaces. Models how intrefaces relate to internal parts
- **Deployment diagrams**
  - A *node* is a communication resource where components are deployed for execution by way of artifacts
  - A *communication path* is an interconnection between nodes to exchange messages, typically used to represent network connections
  - An *artifact* is a physical piece of infromation used in deployment and operation of a system
- **Sequence diagrams**
  - *Lifeline*: represents an individual participant in the interaction
  - *Message*: communication
- **Dynamic modeling** models dynamic aspects of systems: **control** and **synchronization** within an object
  - What are the **state** of the system?
  - Which **events** does the system react to?
  - Which **transitions** are possible?
  - When are **activities** (functions) started and stopped
  - Such models correspond to **transition systems**
    * Also called **state machine** or (variant of) *automata*
- **Starecharts** extend standard state machines in various way
  - *Hierarchy*: nested states used for iterated refinement
  - *Parallelism*: machines are combined via product construction
  - *Time and reactivity*: for modeling reactive systems
- **Summary**
  - Modeling language used to capture different system views
    * *Static*: e.g. classes and their relationships
    * *Dynamic*: state-oriented behavioral description
    * *Functional*: behavioral described by function composition
    * *Traces/collaboration*: showing different interaction scenarios
  - Model are starting point for further phases. But their valusis proportional to their prescriptive and analytic properties
  - Foundation of security analysis and bearer for additional security-related information

Model Driven Security

- **Formal**: has well difined semantics
- **General**: ideas may be specialized in many ways
- **Wide spectrum**: Integrates security into overall design process
- **Tool supported**: Compatible too with UML-based design tools
- **Scales**: Initial experience positive
- Components of **Model Driven Security (MDS)**
  - **Models**:
    - Modeling languages combine security and design languages
    - Models specify security and design aspects
  - **Security Infrastructure**: code + standards conform infrastructure
  - **Transformation**: parameterized by component standard
- **M**odel **D**driven **A**rchitecture
  - A **model** presents a system view useful for conceptual understanding
    - When the model have *semantics*, they constiture formal specifications and can also be used fro analysis and refinement
  - MDA is an **O**bject **M**anagement **G**roup standard
    - *Standard* are political, not scientific, construts
    - They are valuable for building interoperable tools and for the widespread acceptance of tools and notations used
  - MDA is based on standard for:
    - *Modeling*: The UML, for defining graphical view-oriented models of requirements and designs
    - *Metamodeling*: the **M**eta-Object **F**acility, for defining modeling languages, like UML
- **U**nified **M**odeling **L**anguage
  - Family of graphical languages for OO-modeling
  - Wide industrial accpetance and considerable tool support
  - Semantics just for parts. Not yet a *Formal Method*
  - **Class Diagrams**: describe structural aspects of systems. A *class* specifies a set of objects with common *services*, *properties*, and *behaviors*. Services are described by *methods* and *properties* by *attributes* and *associations*
  - **Statecharts**: describe the *behavior* of a system or class in terms of *states* and *events* that cause *state transitions*
- Core UML can be exntended by defining **UML profile**
- A **metamodel** defines the (abstract) syntax of other models
  - Its elements, *metaobjects*, describe *types* of model objects
  - MOF is a standard for defining metamodels
- **Access Control Policies**, specify which subjects have rights to read/write which objects
- **Security policies** can be enforced using a **reference monitor** as protection mechanism; checks whether *authenticated* users are *authorized* to perform actions
- **Access Control**: Two kinds are usually supported
  - **Declarative** $u \in Users$ has $p \in Permissions$: $\iff (u,p) \in AC$
    - Authorization is specified by a relation
  - **Programmatic**: via assertions at relevant program points; system environment provides information needed for decision
  - These two kinds are often conbined
  - **R**ole **B**ased **A**ccess **C**ontrol is a commonly used declarative model
    - *Roles* group *priviledges*
- **Secure UML**
  - *Abstract syntax* defined by a MOF metamodel
  - *Concrete syntax* based on UML and defined with a UML profile
  - Key idea:
    - An access control policy formalizes the permissions to perform **actions** or **(protected) resources**

    - We leave *these* open as **types** whose elements are not fixed
    - Elements specified during combination with design language
  - **Roles** and **Users**
    - Users, Roles, and Groups defined by stereotyped classes
    - Hierarchies defined using inheritance
    - Relations defined using stereotyped associations
  - **Permissions**
    - Modeling permissions require that actiosn and resources have already been defined
    - A permission binds one or more actions to a single resource
    - Specify two relations : Permissions $\Longleftrightarrow$ Action and Actions $\Longleftrightarrow$ Resource
  - Formalizes two kinds of AC decisions
    - **Declarative AC** where decisions depend on **static information**: the assignments of users u and permissions (to actions a) to roles
    - **Programmatic AC** where decisions depend on **dynamic inforamtion**: the satisfaction of authorization constraints in current system state.
- **Generating Security Infrastructure**
  - Decrease burden on programmer
  - Faster adaptation to changing requirements
  - Scales better when porting to different platforms
  - Correctness of generation can be proved, once and for all
- A **controller** defines how a system's behavior may evolve; Definition in terms of *states* and *events*, which cause state transitions
  - Focus: a language for modeling controllers for *multi-tier architectures*
  - *Model view controller* is a common patter for such systems
  - A **statemachine** formalizes the behavior of a controller
  - The statemachine consist of **states** and **transitions**
  - Two state sybtypes:
    - *SubControllerState* refers to sub-controller
    - *ViewState* represents a user interaction
  - A transition is triggered by an *Event* and the assigned *StatemachineAction* is executed during the state transition
- **Dialect** defines *resources* and *actions*

Secure Coding

- **Buffer overflows**
  - A **buffer** is a contiguous region of memory storing data of the same type
  - A **buffer overflow** occurs when data is written past buffer's end
  - They can alter program's data and control flow
  - This is a massive problem and has been so far many years
  - The resulting damage depends on:
    - Where the data spills over to
    - How this memory region is used
    - What modifications are made
- **Layout of virtual memory**
  - *Stack* grows downward and holds
    - Calling parameters
    - Local variables for functions
    - Various address
  - *Heap* grows upwards
    - Dynamically allocated storage generated using `alloc` or `malloc`
- Where would a malicious attacker jump to?
  - Common target: code that creates a (root-)shell
- Where in memory does this code go?

- Exploit code typically placed on the stack
  - Usually, within the very buffer that is overflowed
- Return address must point exactly to the exploit's entry point
  - Non-trivial in practice
  - Trick used of starting exploit with a landind zone of values representing `nop` instructions
- Alternatively, attacker places exploit code:
  - On the *stack*: into parameters or other local variable
  - On the *heap*: into some dynamically allocated memory region
  - Into *environment variables* (on stack)
- A **canary** is a value on the stack whose value is tested before returning
  - It is a random value (hard for attacker to guess) or a value composed of different string terminators
- **Automatic array bounds checking**
  - Compiler automatically adds an explicit check to each array access during code generation
  - Drawbacks
    * It can be difficult to determine the bounds of an array
    * Loss of performance can be substantial
    * Some compilers only check explicit array references
- **Defense programming**
  - Avoid unsafe library functions
  - Always check bounds of array when iterating over them
- **Non executable buffers**
  - Mark stack or heap as being non-executable, thus the attacker cannot run exploit stored in buffers on stack/heap
  - Extend OS with a register string maximal executable address
  - Alternatively, tag pages as (non)executable in the page table
  - Problems and limitations
    * Attacker can still execute code in the text segment
    * Attacker can still violate data integrity
- **Address Space Layout Randomization**
  - Randomizing memory layout
    * Location of stack and heap base in a
    * Order libraries are loaded
    * Even layout within stack frames by compiler
  - Does not eliminate overflow problem
    * Lowers chances of a successful exploit by requiring the attacker to guess locations of relevent areas
- **Format string vulnerabilities**
  - Can crash the program
  - Can read the stacks's constant
  - Can read and overwrite arbitrary memory locations
    * `printf` can modify the contents of memory locations.
- **Unix file system**
  - Directories are hierarchically structured
    * Contents: directories and data files
    * Root of directory tree is the *root directory* `/`
  - User have an associated *current working directory*
  - Each file and directory has an associated **inode** data structure
  - **File descriptor** provide a handle to an inode
- **File name vulnerabilities**
  - Files names are not cononical
  - Dut to links, directory is actually a graph, not a tree
  - File parsing vulnerabilities have bee a problen in past
- **Race conditions** occurs when the results of computation depend on which thread of process is scheduled
  - The result appears to be non-deterministic

  - In reality, the result is determinced by the scheduling algorithm and the environment
- **HTTP** transfers hypertext requests and data between browser and server
  - *Get*: request a web page
  - *Post*: submit data to be processed
  - *Put*: store (upload) sone specific resource
  - On each request, the client sends a *HTTP header* to the server
- **Session management**
  - HTTP is stateless, it does not support sessions
  - Session managments is implemented using **cookies** or **URL query string** to the thread state
- **SQL injection**
  - Input validation attacks where user data is sent to a web server and passed on to back-end system
  - The attacker tries to alter program code on the server
  - SQL servers are standard backends for majority of web servers
  - *Countermeasures*
    * Perform input validation
    * Parse and then substitute, not the other way around
- **Cross site scripting**
  - **Same origin policy** prevents information flow
  - Two pages belong to the same origin iff the *domain name*, *protocol* and *port* are identical
  - **XSS**
    * Web site inadvertently sends malicious script to browser, which interprets the script
    * Script embedded in a dynamically generated page based on unvalidated input from untrustworthy sources
  - **C**ontent **S**ecurity **P**olicy
    * Standard prevents XSS and other code injection attacks
    * Server define white list of trusted content sources

Risk Analysis

- Risk analysis is relevant for all phases of the waterfall
- Identify the most probable **threats** to an organization
- Understand the related **vulnerabilities**
- Relate these to the organizational **assets** and their **valuation**
- Determine **risks** and suitable **countermeasures**
- It's all about *balance*
  - Balancing functional requirements, usability, costs, risks
  - Don't spend 1000 CHF for a firewall to protect 100 CHF worth of data
- Differentialte relevant risks with theorical ones
  - Cryptabalysis of ciphers vs dictionary attacks on password
  - This requires a *proper threat analysis*, i.e., adversarial model
- **Assets**: Things of value to an organization
  - Tangible (physical like hardware or logical like sofware) and intangible
  - Value sometimes difficult to estimate
- **Threat**: Potential cause of an *unwanted event* that may harm the organization and its assets
- **Vulnerability**: A characteristic (include weakness) of an asset that can be exploited by a threat
- Source of threats
  - *Human* with various motives
  - *Nature*
  - *Environment*
  - Not all threats based on a malicious intent
- **Countermeasures**
  - Means to detect, deter, or deny attacks to threatened assets
    * Encryption, authentication
    * Intrusion detection

    * Auditing
  - Countermeasures may have vulnerabilities and are subject to attacks, too
  - Not for free
    * Direct cost
    * Often impact on system function on non-functional behavior
- **Risk** is the possibility to suffer harm or loss
  - Also a measure of failure to counter a threat (you might well choose to ignore certain threats)
  - An organization's risks are a function of:
    * A *loss associated with an event*
    * The *probability/likelihood/frequency of event occurrence*
    * The *degree* to which the *risk outcome can be influenced*
  - Measure *expected loss* resulting from a threat successfully exploiting a vulnerability
- **Risk enablers/vulnerabilities**
  - Software design flaws
  - Software implementation errors
  - System misconfiguration, e.g., firewalls, WLANS, ...
  - Inadequate security policies or enforcement
  - Poor system management
  - Lack of physical protection
  - Lack of employee training
- **Handling risk**: strategies for risk reduction
  - **Avoid** the risk, by changing requirements for security or other system characteristic (followed by redesign/implementation)
  - **Transfer** the risk, by allocating it to other systems, people organization's assets or by buying insurance
  - **Assume** the risk, by either *mitigating/reducing* it with available resources, or simply *accepting* it
- **Risk analysis** is the process of examining a system and its operational context to dertermine possible exposures and the harm they can cause
- **Risk management** involves the identification, selection, and adoption of security measures justified by
  - The identified risks to assets
  - The degree by which the measures reduce these risks to acceptable levels
  - The cost of these measures
- Generic procedure
  - Identify *assets* to be reviewed
  - Ascertain *threats* and the *corresponding vulnerabilities* regarding that asset
  - *Calculate* and *prioritize* the risks; Decide how to *handle* it
  - For assumed risks: Identify and implement *countermeasures* controls, or safeguards - or accept the risk
    * For countermeasures: check that they don't introduce new risks
  - *Monitor* the effectiveness of the controls and assess them
- **Fully quantitative risk analysis**
  - *Goal*: assign independently obtained, objectives, numeric values to all components of a risk analysis
    * Asset value and potential loss
    * Safeguard effectiveness
    * Safeguard cost
    * Probability
  - *Pros*:
    * Effort put into asset value dertermination and risks mitigation
    * Cost/benefit analysis
    * Numbers good for comparisons and communication
  - *Cons*: Costly, accuracy unclear
- **Quantitative risk analysis**
  - *Rational*: Buisinesses want to measure risks in terms of money
  - Difficult for many logical and intangible assets

- Reliance on historical data; nature of future attacks are, in principle, unpredictable
- Problems comparing approximate quantities
- Monetary values give a false impression of precision
- Instead of probability, use categories (high, medium, low)
- *Pros*
    * Simpler as need not determine exact monetary values of assets or probability of different threats succeeding
    * Easy to involve different parties
- *Cons*
    * Even more subjective
    * No single number for decision support
    * No basis for cost-benefit analysis

- **Summary**
  - Risk is a function of assets and threats
    * Value of assets, probability of a threat materializing
    * Existing safeguards
  - Not all threats equally dangerous and countermeasures are not for free; Rely on lists of existing threats and vulnerabilities
  - Most risk analysis procedures rely on some structured means of identifying and evaluating the above items
  - Quantitative assessments are difficult
    * Assignement of probabilities/impact
    * BSI baseline protection on ACTAVE don't even consider probabilities

Threat modeling

- Security engineering is not yet a mature discipline
- Safety engineering and associated methods are better established
  - *Failure modes and effects analysis*: bottom-up, textual
  - *Fault Tree Analysis*: top-down, graphical
- Methods may be used individually or in combination
- **F**ailure **M**ode and **E**ffect **A**nalysis, also called FMECA, to emphasize **C**riticality
  - *Goal*: identify possible root causes of faults early
  - Bottom-up (inductive) addressing system, design, and process
  - Choose component/subsystem/parts and analyze possible failures and their effect on the rest of the system
  - Often used in mechanical or hardware-oriented systems
  - Consider each part of the entire system
  - How may it or its subsystem fail?
    * **Fault**: inability to function or an undesired functionality
    * **Failure**: occurrence of a fault
    * **Failure mode**: manner which fault occurs
  - Analysis based on *historical data*, expert opinion
  - Rank failure mode
    * **Occurence**: relative probability of malfunctions occuring
    * **Severity**: Relative severity of worst possible outcome
    * **(Non) Detectability**: Probability that failure will be detected/corrected
    * $\mathit{Criticality} = \mathit{Occurence} \times \mathit{Severity}$
    * **R**isk **P**riority **N**umber $= \mathit{Occurence} \times \mathit{Severity} \times \mathit{Detectability}$
  - *Procedure*
    1. Define system to be analyzed
    2. Construct block diagrams of systems
    3. Identify potential item and interface definitions
    4. Evaluate and rank failures

    5. Identify possible causes and appropriate actions
    6. Dorrective design: take actions to eliminate/reduce high-risks FMs
    7. Documentation
  - It appears to work well in practice
  - Like all other techniques, no guarantees
    * Garbage in ⇒ Garbage out
    * Easy to overlook human error, effect of hostile environments
  - *Prerequisites*
    * Understand how the system works
    * Done when design/architectures is available
    * Experience with possible problems
  - Leads to facilitated discussion with different group members
  - Unique to FMEA: probability of detection (relevant for security)
- **Fault tree analysis**
  - *Goal* identify conditions leading to system failure (top level event)
  - Aims at finding the **sources** of a system failure
  - Deductive top-down method
  - Quantitative and qualitative
  - Graphical representation of causal relationships
  - *Prerequisites*
    * Undrestand how the system works
    * Done when design/architecture is available
    * Facilitated discussions with different group members
  - *Procedure*
    * Input: plan of the system and FMEA, if existent
    * Determine undesired (top-level) events
    * Identify event(s) that lead to the top-level event
    * Leaves: possible causes
    * Symbols used: AND/OR
    * Identify cut sets: those events that together lead to system failure
    * Perform quantitative or qualitative analysis on resulting tree
  - **Primary event** (leaves)
    * *Basic events*: no precursor; probabilistic
    * *Unddeveloped events*: no major effect alone on the system
    * *External events*: expected to happen; not a fault
  - **Intermediate events**: Link primary of intermediate events via AND/OR gates
  - **Expanded events**: Need a separate fault tree to explain
  - A **cut** is a set of event that, taken together, lead to the top level event
  - A **minimal cut** is a cut that is no longer a cut if an element is removed
  - In general, there are many cuts and many minimal cuts
- **FMEA** may indicate conditions that are not controllable by the system, e.g., security breach at a particular point
  - This gives rise to new requirements for the involved components
  - Closest analogy is *data pathways* where we aggregate requirements as we move from classes to components to systems
- **TFA** starts with an undesired top-level event representing a violated secutity requirement
  - Decomposing it into possible causes may lead to lower-level causes and suggest new security requirements
  - This is the basis of attack trees
- **Safety**: failures arise from **faults** occuring
- **Security**: *failures* are the **unwanted events** that occur when a **threat agent** materializes a **threat** through an **attack** that exploits a **vulnerability** in the system
- **Attack trees**
  - Nodes are attacks (threats)

- Top level goal may be obtained from misuse cases
- Refine as needed
    - Alternative attacks
    - Composite attacks
- Assign attributes to nodes
    - Probabilities
    - Estimated impact
    - Compute probabilities, impact, or risk of cut sets
- Use structure of system and environment

- **Summary**
  - System design models help to identify threats
  - Data pathways as means to
    - Identify critical system parts and
    - Support risk assessment
  - Use of attack trees to capture threats

Security Design

- **Safeguards and countermeasures**
  - **Avoidance controls**
    - Safeguard used to proactively minimize risk of exploits
    - Either reduce their likelihood or impact
  - **Assurance**: Tools and strategies to ensure the effectiveness of existing controls and safeguards
  - **Detection**
    - Technique and programs to ensure early detection interception, and response to security breaches
    - Virus scanner and audits
  - **Recovery**
    - Planning and response services to rapidly restore a secure environment and investigate sources of a security breach
- **Security design principles**
  - Use proven patterns and principles
    - Avoid security by obscurity
    - Least priviledge
    - Keep security mechanism simple
    - Defense in depth
    - Detect intrusions
  - Use **standard** and **best practices**
  - **Consistent security level**: define a baseline and enforce it everywhere
  - **Take appropriate measures at right level**
    - Adopt software where needed ratehr than hacking infrastructure
    - Use capabilities provided by the given technologies rather then employing additional components
  - **Use mature libraries**
  - **Use proprietary solutions as a last resort**
    - Standard solution are usually more secure, efficient, and robust than home-grown ones
    - Maintenance ?
  - **Generate implementation to avoid programming errors**
    - Access control is a good candidate
    - Require high-quality generators
- **Countermeasures** can be categorized as follows:
  1. Integrate or configure existing security mechanisms (utilize systen security mechanism)
  2. Implement security functions in application logic
  3. Refactor the software architecture (Reduce “attack surface”)

1. **Security mechanism integration**

- Integrate, configure, or employ existing mechanisms
  - Network: firewalls, VPNs, network-authentication and access control
  - OS: hardening, file system access control
  - Application/webserver: transport encryption, authentication, access control
- *Pros*
  - Baseline protection
  - No application support required, so good for black-box integration
  - Usually well-undertsood by IT departments and administrators
- *Cons*
  - Application layer attacks cannot be stopped at the network level
  - Nontrivial

2. **Implement security in the application**

- Some countermeasures best implement within application
  - I/O validation and application-specific checks
  - Application-level encryption
  - Access control
- Requires that security design is integrated into the development process
- Use of standard security APIs and modules is advisable
- *Pros*
  - Best fit to application
  - No additional system components, reduces costs for licensing and operation
- *Cons*
  - Expensive
  - Error prone, test intensive

3. **Refactor software architecture**

- Improve/simplify security design
  - Reduce attack surface
  - Split systems into highly-critical and less-critical parts
  - Allocate security-sensitive functions to toher systems
- *Pros*
  - Simplifies security design
  - Cheaper and less error-prone
  - Sometime the only way to achieve security goal
- *Cons*
  - Significant impact on overall project
  - May have negative impact on other functions
  - Causes strongest resistance and conflicts
- **Summary**
- Risk can be mitigated, transfered, or accepted
- Mitigation based on different safeguards
- Security design is a creative process, guided by principles
- Measures fall into different categories
- Threat model and risk analysis must be synchronized with design
- Code generation can improve quality of security implementations

Code scanning

- **Code scanning** is a verification thechnique
  - Take code as input
  - Supplements manual code inspection

    - Is a countermeasure against implementation problems
- *Problem categories* (from security perspective)
    - **Input validation and representation**: Buffer overflowm injection attacks, etc
    - **API abuse**: abuse contract between and callee; provide wring input or make too strong assumptions about output
    - **Security features** e.g., don't hardcode password in source code
    - **Time and state**: e.g., race conditions
    - **Error handling**: handle errors poorly or not at all
    - **Code quality**: dereference null pointers, infinite loops, etc
    - **Encapsulation**: lack thereof
- Code scanning = pragmatic static analysis
- **Flaws** concern problematic behavior
    - **Failures**: derivation of bahavior detectable at system interface
    - **Errors**: deviation of system's behavior from intended one
- Behavior properties in general undecidable
    - Termination, reachability of a program point
- Tools must therefore
    - *Not always terminate*, or
    - *Over-approximate* behavior (returning *false positives*), or
    - *Under-approximate* behaviors (returning *false negatives*)
- **Semantic analysis and structural rules**
    - Build symbol table along with AST
    - This can be used for type checking, which helps find bugs
    - Types also help in specifying *structural rules* for bug finding
- Tools build **control flow graph** on top of AST
    - **Basic block**: sequence of instructions that is always execurted, i.e., no jumps in/out of middle, no branching
    - **Forward edges**: potential control flow paths between BBs
    - **Backward edges**: possible loops
- **Dataflow analysis**
    - Determine how data moves through program
        * Traverse control flow graph and note where data generated and used
    - Implementation trick: convert function to static Single Assignement
        * Can assign to a variable only once, so make unique with indicies
        * Make it trivial to determine where value comes from
    - Simple compiler application: constant propagation
- **Taint analysis**
    - *Source rules* defines locations where tainted data enters system
    - *Sink rules* define locations that should not receive tainted data
    - *Pass-through rules* define how a function manipulates tainted data
    - *Clense rules* special pass-through rules that remove taint
- **Input validation**
    - *Attack surface*: all places where program accepts input
- **Summary**
    - Code scanning should play a central role in code review
        * Aids understanding code
        * Helps finding common bugs
    - Surprisingly effective
        * Everyone makes dumb mistakes
        * Simple patterns can describe remarkable subtle bugs
        * Threads/crypto/... more complex than most people think
    - Pragmatic, conservative, static analysis techniques play a major role
        * Augmetnted with lots of domain knowledge

Security Testing

- **Validation and Verification** (V&V) evaluates the *quality* of software with respect to its *specification* and the overall system *requirements*
- Target: *Design flaws*:
  - **Formal methods** (model checking, theorem proving): check the design with respect to the specification
- Target: *Implementation flaws*:
  - **Static analysis**: reason about programs without executing them: manual or automated code inspection
  - **Dynamic analysis**: reason about programs by executing them: *testing*, and run-time analysis
- Tests are attempted refutation
- *Black box testing*: Programs map inputs to outputs, but we do not know how
- *Limitations of testing*:
  - Observations are finite
  - Program's input domains are often infinite, while testing amounts to executing a program on finitely many inputs
    * Thus, testing *cannot refute* $P \vdash \theta$, whith an *existential* θ

$$\exists e \in Executions(P) \quad \phi(e)$$

    * Consequently, testing *cannot verify* $P \vdash \theta$ for a *universal* θ

$$\forall e \in Executions(P) \quad \phi(e)$$

    * $\vdash$ means satisfy where P is a program and θ its specifications
- Testing aims at **refuting** the hypothesis that a system satisfy a specification
- Testing is confined to choosing finitely many inputs, and observing the system's behavior for a finite length of time
- The finite nature of testing entails that certain specifications are *irrefutable* through tests. Other verification techniques such as static analysis, must be used then
- **Test Selection Problem**: Which inputs to choose for testing
  - A test generation method is a *systematic* approach to test selection, ideally amenable to automation
  - Test generation methods can be classified as
    * *Random*, requires nothing
    * *Fault based*, requires a fault model for P
    * *Model-based*, requires a formal model of P
    * *Specification-based*, requires a formal model of θ
  - *Random testing* is the base-line: Any non worthy test generation method should outperform random testing in terms or relevant **failures**
- **Fault Model**
  - **Failure**: a deviation of P from the expected observable behavior
  - **Fault**: the cause of the failure
- Testing reveals *failures*
- Finding and fixing the inderlying *faults* is **debugging**
- A **fault model** describe a class of (common) faults
- Testing is searching for failures. The search should ideally be conducted in the light of how faults come about, which suggests how failures are likely found
- **Fault models** capture repeated programming mistakes
  - Each fault model reflects a small number of mistakes
  - **Interaction Rule**: most failures are inducted by single factor faults or by jount combinatorial effect (interaction) or two factors, with progressively fewer failures induced by interactions between three or more factors
  - *Recipe*:
    * Identify the input domain
    * Choose a suitable fault model for the input domain

      - Partition the input domain using the fault model
      - Select representative test inputs per partition
      - Test Orable: system's specification
- **Test adequacy Criteria**
  - We select a *finite* subset S of the *infinite* set D of inputs
  - **Specification coverage**: **Adequacy** of $S =$ the percentage of the specification obligations and prohibitions that are exercised by at least one test in S
  - **Model coverage**: **Adequacy** of $S =$ the percertage of the models components that are exercised by at least one test in S
- **Coverage** and **mutation analysis** are widespread adequacy measures. They both however rely on hypotheses
  - The relationship between coverage/mutations anlysis and **failure detection** must be empirically validated
- **Requirements** are about **resources**
  - We reduce requirements to **specifications** for **systems**
  - The **reduction** relies on appropriate environment **assumptions**
- Two types of **Security tests**:
  - **S-Tests**: Refute $System \vdash SPEC$, e.g. using fault-based tests
    - *Independant* from the adversary
  - **E-Tests**: Refute $(Environment||System||adversary) \vdash EA$
    - *Depend* on the adversary model
- *Adversary model* itself is not subject to tests
- **Adequacy** of *S-Tests*: functional adequacy measures, such as *coverage* and *mutation analysis*
- **Adequacy** of *E-Tests*: Ideal: the validity of each environmental assumption is "adequatly" tested
- *Challenges*:
  - EAs are hard to explicit
  - Hard to say how well a CWA is tested
- $EA, SPEC \Rightarrow REQ$
- **Security flaws** revealed throught security tests:
  - System fails to satisfy SPEC
    - Revealed through *S-Tests*
    - Debugging: Fix the system
  - EA is violated:
    - Revealed through *E-Tests*
    - Debugging: Fixing the system falls short. Fix the design, update the security rational
  - Security testing does not account for flawas rooted in unelicted requirements or weak attacker models
- **Summary**
  - **Security testing**'s goal is to invalidate the *requirement*: the protected resources cannot be accessed by unauthorized entities
  - **Security rationale** support decomposing the requirements into a system *specification* and an environmental *assumption*
  - **S-Tests** goad is to refute the hypothesis that the system satisfy its *specification*. Functional testing methods and tools apply here
    - Security testing > testing the system w.r.t. its (security) specification
  - **E-Tests** goal is to refute the hypothesis that an *assumption* is valid in the system's environment in the presence of an *adversary*
  - **E-Tests** are hard to generate because environment do not admit delimitation, and environmental *assumptions* are hard to explicate