Security Engineering fiche

Pierre Colson

December 2022

Contents

Introduction **1**

Requirements Engineering **2**

Modeling **2**

Model Driven Security **4**

Secure Coding **6**

Risk Analysis **8**

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 precisly 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 beteween 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 $\iff$ Action and Actions $\iff$ 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 occurence*
        - 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