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 bahavior 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*

Summary

- **Introduction**
  - 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**
  - 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**
  - 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