summary

54663 characters in 7590 words on 1502 lines

Florian Moser

December 18, 2019

1 Security Engineering

1.1 composed of

Software Engineering (systematic approach to dev/ops/maintenance)
Information Security (reduce risks to information assets)

1.2 target

building secure systems in the face of error/malice
provide tools, processes & methods
for design, implementation, testing and maintainenance

1.3 glossary

CIA
confidentiality (no improper disclosure of information)
integrity (no improper modification)
availability (no improper impairment of service)

Murphys Computer
computer working against the system
helps to design secure system

1.4 challenges with systems

integrate communication systems, procedures, people
cost pressure forces feature-oriented development
composed of self-written & commercial components

1.4.1 general challenges

complexity
most complex man-made artifacts
many million lines of code in a big product
many different possible states (2 integers = grains of sand)

young discipline
no fixed way to solve many problems
reinvented quickly

1.4.2 security challenges

security not continuous
usually minor oversights enough for failure (failed checks, incorrect config)
more dramatic impact than if a functionality is broken

workflows with security as addon
adding features is rewarding
but CIA which restricts behaviour
done in the end leading to quircky design

hackers are not typical users
hackers explicitly try to enter an unsafe / insecure state
hard to think about all possible reacheable states

side channels
hacker may use tools outside the system

1.4.3 implications

high defect rate
by 1000 LoC around 50-60 bugs arise
reduce with modern design (-50%) & reviews (-95%)

2 development modes

2.1 classic ("code-and-fix")

write full system & release
fix what is broken

evluation
not transparent because no checkpoints
hard to maintain security
hard to adapt to end-user needs

2.2 variant ("penetrate-and-patch")

release system
bugs then disclosed by users/pen testers to developers

evaluation
after bugfix need to distribute patch rapidly
bugs can be abused

2.3 first process model ("phase model")

also called System Development Life-Cycle, Waterfall
1968 manifest for software development sponsored by NATO
decomposed into phases which each produce a product/artifact

evaluation
clear deliverables after each process
predictable scheduling, budgeting, personel requirements & result
but assumes requirements fixed, abstract design feasible
hides conceptual, technological & personnel risks
risky big bang delivery (testing, feedback, integration at the end)

improvements
add feedback loop (but removes checkpoints, makes scheduling hard)

2.4 V-Model

more flexible & complex waterfall variant
ISO standard which regulates activities and their relationships
used by military/administrative projects in DE

structure
form of a V, start is top left, end is top right
down flows specification, up bubble artifacts
left sends specification to right
right sends feedback to left

levels (left / right)
system requirements analysis / transition to utilization
system design / system integration
HW & SW requirements & design / SW implementation (component level)
detailed SW design / SW implementation (module/database level)

2.5 Rational Unified Process

composed of phases which include multiple waterfall phases
each RUF phase is composed out of multiple iterations

2.5.1 evaluation

reduces risk by priorizing & building PoC
encourages all stakeholders to participate

2.5.2 phases

inception
rough system definition for initial cost
create basic use cases, project plan & initial business risk analysis

elaboration
decide go/redesign/cancel project
mitigate high-risk, detail use-cases & architecture, create dev plan

contruction
create the system
build & elaborate features, release

transition
operate the system
improve system, beta test, validate, training

2.6 agile methods

lightweight processes which put software first
small teams, small increments including all phases
stresses face-to-face communication & customer interaction
continuously updates risk analysis
uses devops to avoid conflicts with stability vs progress tradeoff

2.7 ressources

microsoft security development lifecycle
OWASP Secure Software Development Lifecycle Project(S-SDLC)

3 waterfall

composed out of 5 clearly defined phases
other methodologies may order phases differently
but work done in these phases is needed in all projects
hence relevant to be analysed

3.1 requirements engineering

elicit, analyze and document system requirements with shareholders
build models to make requirements precise
do risk analysis, priority list
check feasability / business cases

security requirements
identify data criticality
authorization
valid/invalid use cases

tasks / output
requirement analysis (feasibility study, requirements sketch)
requirement definition (functional / non-functional requirements)

3.2 design

fix system-architecture (software & hardware requirements)
recursively decompose in subsystems & define their interactions

security models & patterns
define encryptions
access control, key-management
logging

tasks / output
system spec (specification, test plan, user docu)
architecture spec (specification, system test plan)
interface spec (specification, integration test plan)

3.3 implementation

develop subsystems, further design algorithms / data structures
reuse existing products

secure coding
use secure languages, defensive coding, guidelines
do reviews & analysis

tasks / output
finished program (detailed specification, unit test plan)

3.4 validation & verification

reviews, static analysis
interactive program verification
blackbox/whitebox, regression testing
unit, module, integration, system, acceptance testing

security leak detection
vulnerability detection (static/dynamic analysis)
risk-based & penetration testing

tasks / output
unit, module, system tests (report)
integration tests (report, final user docu)
acceptance tests (report, documentation)

3.5 operation & maintenance

installation, patching, dependency & functional updates
bug tracking, backup, recovery, continuity & capacity planing
user education, help desk & emergency response

secure distribution & monitoring
distribution of code/patches/keys/config
monitor system activity for deviations

results
running stable product

4 requirements engineering

concerned with functions & constraints on software systems

4.1 requirements

specifies effects & non-effects under assumptions in a problem domain
includes development process, data exchange, deployment, documentation
difficult because imprecise, incomplete, changing
hard to imagine all possible ways to use system (including by adversary!)

kinds
functional requirements for purpose(use cases & goal-oriented)
non-functional requirements for quality (security, performance, usability)

targets
quantifiable and precise (because need to verify implementation)
consistent & priorized (because requirements often conflict)

4.2 activities

elicitation
determine requirements with stakeholders
combine strategic objectives, domain knowledge, organizational context
perform interviews/workshops, observe processes, create scenarios/models

analyse
ensure requirements are clear, consistent & complete
build models of system
do business & security risk analysis
gather implicit requirements (not asked for but needed)
like view before edit, admin roles & their permissions
choose functional requirements to implement non-functionals
like to use IPSec to ensure traffic is secure

specification
document desired behaviours for schedule & cost planning
emphasis on what, not how (which is answered in dev phase)
may include precise models or formal specifications
may be structured according to standards like IEEE 830-1984

validation
ensure the right system is built
build prototypes, informal/formal models
include changed requirements of stakeholders

4.3 security requirements

formalize creation & usage (what, when, who)
motivated by adversarial context
conflict with cost, usability, performance

consider
CIA (confidentiality, integrity, availability)
user-related (identification, authentication, authorization)
misuse (intrusion detection, surviveability, physical protection)
auditing (non-repudiation, anonymity, data protection)
external events (earthquakes, floods)
internal events (networks, systems, people)

sources
laws / sector specific provisions (GDPR, FDA, credit card regulations)
guidelines (algorithms, technologies, password complexity)
domain/application specifics (like auditability for admins)

4.4 security policies

data criticality
"high-water mark" as recommended concept
define for each system & dimension (usually CIA)
"low" when minor damage and primary functions OK
"middle" when significant and heavily impacted
"high" when major (life-threatening) & impossible
document as table or include in class diagram

authorization
defined for each information (like name), role
concerning read (confidentiality) & write/excute (integrity)
"least priviledge" as recommended policy
values could be are "own", "all", "no"

4.5 IEEE guide

functionality (what system should do)
external interfaces (UI, systems, hardware)
performance (speed, availability, response time, recovery time)
attributes (portability, correctness, maintainability, security)
implementation design contraints (language, resources, OS, standards)

structure
introduction (purpose, main scope, definitions, references)
general description (perspective, function, user types, general constraints, dependencies)
external interfaces (dependencies, other systems, hardware)
functional requirements (description, priority & specifics for each feature)
non-functional requirements (security, maintainability, ...)

4.6 example

payment card indistry
enforces consistent security & data protection measures
organisational (policies, prodedures) constraints
technical (network architecture, software design) constraints
certified vendors successfully implement these requirements

5 modelling

bridges the gap between software requirements and design
to specify requirements as precisely as possible

5.1 modeling languages types

multiple languages with different focus combined in practice

focus on system view
static (single point in time)
dynamic (how static evolves)
functional (describes actual function)

religion
object oriented (OO) like UML, OOA/OOD
algebraic specification

5.2 model

construction or mathematical object to describe system

5.2.1 entity relationship modeling (E/R)

developed for data-centric applications
define sets of data and their relationships
implemented on database level (tables & relations)

visualization
entites as rectangles (person)
attributes as ovals (first_name)
relations as line with diamond between entities (is_related_to)
unary relations (subset) like relation (is_disabled)

evaluation
simple, successful in practice & tool support
but not standardized
but need extensions for complex relations (>2, restrictions)

5.2.2 data flow diagrams (DFD)

for functions and data flows
useful for requirements plan and system definition

visualization
input as rectangle (request_book_by_user)
function as circle (find_book)
data store as border top/bottom (book_db)
data flow as arrow (shelve_number)
output as oval rectangle (show_book_booked)

semantics
"read" as arrow from datastore to function
"save" as arrow from function to datastore
suggest additional semantics by function names

receipe
start with informal high level (just input, output)
continuously decompose into functions, data stores, ...
create for different hierarchies

tool support
CASE tools and similar

6 UML

unification from various previous developed methods
add precise semantics & domain specific semantics
14 different views of the system (both static & dynamic)
intuitiv, tool-support and de-facto standard
but informal, partly cryptical & unstable specification

6.1 object constraint language (OCL)

first order logic for invariants, pre & post conditions
provides functions for constraining the class diagram

keywords
context to refer to class
self to access current context object
inv to declare invariants

examples
Webpage.allInstances() to get all instances
self.articles→select(c |c.name = 'Name')→size()
Webpage.allInstances()→forAll(p|Webpage.allInstances()→forAll(q |p<>q implies p.articles→intersection(q.articles)→isEmpty()))

6.2 use cases

specifies who can do what with the system
can extend with new stereotypes

6.2.1 elements

users
add types of users and "attacker"/"unlucky"
define their inheritance by pointing to generalization

use-case
use cases of respective users
potentially prohibided/misued functionality from atttacker users
add "extensions points" with variants used by conditions

conditions
can add to "extend" relation
specify name of extension point from target bubble
specify condition to activate (like "user not logged in")

"<<includes>>" relations
preconditions (buy → login)
for example "edit account" - "include" → "login"

"<<extend>>" relations
refined use-cases (SSO → login)
for example "SSO" - "extend" → "login"

"+prevents"/"+detects" relations
defenses for attacks
"use HTTPS" - "prevent" → "eavesdropping"

6.2.2 visualization

users as stickmen; with black head if malicious
inheritance as arrow with white head to more general
use-cases as bubbles, connected to users with lines
relations as dashed lines
conditions as point on relation with bubble describing what

6.2.3 relevance for security engineering

authorization policy with roles as actor
system boundary as system interface
useful to derive misuse cases

6.3 activity diagrams

despict the sequence and conditions for coordinating activities
visualize system parts & dependencies

elements
action (single no futher decomposable step)
activity (flow of activies and actions)
control flow (edges ordering activies)
decision (change flow taken based on goards or unite them)
object flow (edge with data passed along)

visualization
input as black dot, output as black dot with circle around
action as oval rectangle
type as rectangle, text like cart: ShoppingCart
activity as oval rectangle with inner flow
control flow as arrows

decision as 45 °rectangles with optional guard text on output
object flow as text close to arrows

relevance for security engineering
understand application/data flows, participants
informs authorization policy

6.4 class diagrams

shows a systems classes, their attributes & relationships

6.4.1 elements

class (object with same features, constraints, semantics)
attribute (structural feature of class)
operation (behavioral feature of class with calling infos)
calling infos (include name, type, parameters, invocation constraints)

association
does_something relation
generalization (is-a relation, specific to more general)
n-ary aggregation (season as combination of player, year, team)
aggregation (is-part-of, other can exist independently)
composition (other has no independent existence)
association class (add class properties to relation)

6.4.2 visualization

class as rectangle; first row in rectangle
attributes as second row in class rectangle
operations as third row in class rectangle
specific responsibilities as text in fourth row in class rectangle

association
as line with semantic text like +owner
generalization as arrow
n-ary aggregation as lines meeting in empty diamond
aggregation with empty diamond at whole
composition with black diamond at whole
association class as class with line to association

6.4.3 annotations

can be added to associations

multiplicity
constrains the relation by number (like 2) or range (like 0..∗)
{set}for unordered, unique (default)
{ordered}for ordered, unique
{list}for ordered, duplicates
{bag}for unordering, duplicates
"company employs 6 persons" → 6 written close to person

navigation
restricts query/operation direction by arrow (to enable) or cross (to deny)
explicit notation requires all (else unspecified)
implicit notation assumes most navigatable possible
"student has marks" → cross at "student", arrow at "marks"

role
no semantic consequence, helps to describe association
"owner names pet" → "owner" written next to "person"

directed name
arrow which specifies in which direction text is implied
"owner names pet" → arrow after "names" pointing to pet

6.4.4 security engineering relevance

informs authorization policy and CIA

6.5 component diagrams

model system components

elements
component (encapsulates its functionality in the environment)
provided interfaces (interfaces implemented by component)
required interfaces (interfaces required to fulfil functionality)
assembly connector (to link interfaces of different components)
ports (connects interface with inner components)

visualization
components as rectangle with inner components
provided interfaces as outside line with half-circle at the end
required interfaces as outside with filled circles at the end
interfaces might be named
assembly connectors as half-circle attached to filled circle
port as square where interface line meets component
port might be connected to inner component

security engineering relevance
analyse data flow
detect security relevant components/applications

6.6 deployment diagrams

model execution architecture of system

elements
node (<<system>>)
communication path (connect nodes using a +technology)
artifacts (files which <<deploy>>on node)
components (which <<manifest>>artifacts)

visualization
node as rectangle cubes
communication paths as lines with transport mode as text
artifacts as rectangles
components like from component diagram

example
three tier systems connected with http, grpc
browser deploys to users PC, binary to server, SQL to db server
database component manifests sql for db server

security engineering relevance
check reachability from internal/external systems
assign CIA to paths & do impact analysis

6.7 sequence diagram

describe single interaction
focus on exchanged messages & show active processes needed

elements
objects with lifetimes
requests (target, method name, content)
response (content)

visualization
objects as rectangles on top x axis
lifetimes as vertical lines down from each rectangle
non-livetimes as dotted vertical line (browser, inactive)
requests as arrows with text above for specification
responses as dotted arrows with text above for content

6.8 statecharts (dynamic modeling)

extension to state machines
adds hierarchy, parallelism and time & reactivity

6.8.1 elements

class diagram with receivable events
transitions caused by events under conditions
transitions may execute actions
choice points to create multiple paths with potential conditions

states
simple
composite which contain own start/stop
concurrent to synchronize concurrent stream
pseudo like initial/final, fork/join

6.8.2 visualization

receivable events as <<event>>stereotype
^otherSM as events, methodName as actions, [conditions]
actions written after / (like [else] / invalid++)

states
simple as rounded rectangles
composite as state around smaller states
concurrent like composite with dashed line inbetween streams
fork/join as bold lines, all paths must visited

6.9 UML extensibility

build domain-specific languages in UML

stereotype
define new type for UML with name, color, icon
extends existing UML type and defines tagged values
can be applied to model with the <<Base>>stereotype
for example Event with <<Persistent>>declares that an event is stored

tagged value
additional model attributes (name-value pair)

constraint
text enforcing additional constraits

7 model driven security

get from requirements to implementation
get from security policies to design models

7.1 objectives

formal (mathematical semantics)
general (many ways to specialize idea)
usable (reuses familiar concepts & notation)
wide spectrum (integrates security into design process)
tool support (UML-based design)
scalable (academic & scientific usage)

7.2 background

model driven architecture (MDA)
an emerging standard of OMG
interopable because clearly defined

MOF
standard to define metamodels (M3)
metamodels to define modeling language (M2)
modeling language to define application (M1)
application to define instance (M0)

7.3 process

choose MDA with security extension
create system model (class diagrams & relations)
combine with security model
use MDA to transform model to target system

7.4 security policies

define confidentiality (view) and integrity (save) of data
can be formalized as access control policies
declarative (permissions) and programmatic (state) often combined

role based access control (RBAC)
decouples users/permission by roles
combine with role and/or user hierarchy
combine with stateful contraints

7.5 secureUML

abstraction of RBAC combined with action/resource abstraction

7.5.1 structure

subject
composed of group/user
can inherit to group

role
assigned to subjects
can inherit from itself

permission
assigned to roles
potentially has authorization contraints
on state, method arguments, global system properties

action
assigned to permissions
represents something to do with a resource (read, execute, ...)
composed of atomicAction or compositeAction
can inherit to compositeAction (fullAccess → read) and from itself

ressource
assigned to action
can inherit from itself
represents something protected (like State, Action, Class, ...)

7.5.2 application

combine it with class diagram (for example using ComponentUML)

example
user has permissions for meeting
permissions represented as an association class
permissions assign actions to model anchor or subelement
potential actions include read which allows user to read element
actions are potentially restricted by pure authorization constraints

7.5.3 formalizations

combines declarative/programmatic AC to decide if access is granted
in first order logic (variabes, functions, predicates)
for q $element_{of}$ Q, u $element_{of}$ Users, a $element_{of}$ Actions it must hold q |= AC(u, a)

declarative AC
static information which user can execute action
variabes are SecureUML elements (users, ..., actions)
functions are not needed
predicates are if in relations (UA user ↔ role, ..., AA permission ↔ action)
and if in hierachy (subjects inheritance, ..., actions inheritance)
there is r, p such that UA(u,r) PA(r,p) AA(p,a)
then include hierarchies for subjects, roles and actions

programmatic AC
dynamic information which restrictions apply to permissions
formulated as OCL formulas
variables are all classes in system model
functions for all attributes, pure methods, n-1 associations
predicates for all m-n associations

7.6 componentUML

class based language for data modeling

components
<<entity>>with attributes & methods
<<enumeration>>like an enum
unary/binary associations between entites

7.7 dialect

thing in between two modeling languages
ideally combine one for system design and one for security

7.7.1 combine syntax

merge abstract syntax by combining metamodels
merge notation and define well-formedness of OCL rules
identify protected resources
identify resource actions
define action hierarchy

7.7.2 example ComponentUML & SecureUML

identify resources
using subtyping
hence all componentUML elements (Entity, EntityMethod, ...)

identify actions
using named dependencies
hence create arrows from resource types to action classes
create/delete to atomic actions
fullAccess, read to composite actions

7.7.3 semantics

system should behave as before the combination
except that certain actions may be restricted
can define transition relation R with all valid transitions
trace is valid iff (s_1, a_1, s_2) in R

example SecureUML
combine system states q with RBAC states q2
{((q2, q), a, (q2, q')) |(q, a, q') $element_{of}$ R <q2, q>|= AC(u, a)}

example ComponentUML + SecureUML
actions formalized like (set_a, e, v) (set variable a of entity e to value v)

7.8 generate security infrastructure

better maintainability, portability to other platforms, provable correctness

EJB
declarative AC as XML, flattened because EJB knows no hierarchies
programmatic AC as Java, throwing exception if fails

.NET
same than EJB, but instead of XML can use attributes

7.9 controller

defines system behaviour
states & events which cause state transactions

multi-tier architecture
visualization (views data; like browser)
persistence (where model is stored; like DB)
controller (control flow & data flow between visualization/persistence)

metamodel (MOF)
controller associated to subcontroller & statemaschine
statemaschine associated to states
state consist of viewState or subControllerState
state associated with transitions
transitions may execute statemachineActions
transition triggered by an event

meeting example
mainController has viewState ListMeeting
contains cancel transition which executes cancelMeeting action
contains create transition which changes viewState to CreateMeeting

security UML dialect
combine with SecureUML by making controller/states (recursively) activatable
when activating a controller, its actions can be executed if authorized
when activating a controller recursively, all its actions can be executed
when activating a state, its substates can be entered if authorized
when activating a state recursively, all its substates can be entered

secureUML dialect example
permission table for each user / controller
permissions define who can activate controller or execute actions

8 secure coding

8.1 implementation

may deviates from the design (both good and bad)
may needs to define unspecified properties (password length)
may introduces new types of vulnerabilities

8.2 C workings

pointer points to address; contents can be read out
can override pointer when not careful
illegal instruction thrown when return address invalid
segmentation fault thrown when writes to inaccessible memory

stack frame layout
m - 0th parameters (0th parameters first because may contain #parameters)
return address
return value
ebp (base pointer of caller)
0th - n local variable

virtual memory layout
stack growns down by decreasing stack pointer
heap grows upward by allocating using alloc malloc
data contains statically allocated storage
text contains executable code (read only)

funtion call
push arguments on stack (esp increased automatically)
push ebp, then replace ebp with esp
increment esp for local variables of function
do processing of function
leave to pop ebp, esp
ret to jump to next instruction, decrement esp

assembly details
call adds the return address & increments esp

8.3 buffer overflows

buffer is continuous area on memory
overflow occurrs when data written past buffer end
>15% of serious vulnerabilities

examples
morris worm propagated itself using buffer overflows
mozilla heap buffer overflow in GIF viewer allowed RCE

affected
local/web applications
browsers & plugins
OS and protocol stacks
implementations of crypto algos
firewalls, type-safe language interpreters

8.4 exploit examples

8.4.1 gets overflow

attacker passed too long input
can change data & control flow

gets background
gets fills buffer from bottom to top
stops when read string is terminated

attack
gets overrides stack frame if allocated buffer too small
can change return value (change data integrity)
can change return address (change control flow)

code defense
create and check canary value
use only safe functions
automatic array boundes checking (but slow, difficult)

architectural defense
prevent execution of heap/stack
adress space layout randomization (heap/stack location, library loading order)
but still data intergrity danger

8.4.2 printf format string

attacker modifies used format string
can expose secrets, crash program or modify

format string background
has placeholders starting with % to print special stuff
each placeholder represents one argument on the stack
%i prints out integer (read)
%s prints out until \0 reached (unrestricted read)
%n stores #already printed chars in passed location (write)

attack
printf does not detect if too many placeholders passed
then reads out places further up in stack than supposed to

defenses
know exactly how library functionality works
scan for potential vulnerabilities
use weaker functions

8.4.3 SQL injection

attacker modifies used SQL query

attack
pass "' OR '1' = '1'" to get a valid OR
pass "abc'; —" to finish statement, quote the rest

defenses
separate control/data channel (prepared queries)

8.4.4 phone phreaking

attacker uses control frequences of provider

attacker
specific frequences cause landline calls not to be billed

defenses
separate control/data channel

8.5 web applications

widely deployed
general OS/network defenses not enough

8.5.1 background

HTTP overview
URL + optional arguments accessed by client
header contains language, character encoding, browser
GET request has URL & headers
PUT/POST additionally has body
TRACE request advises server to mirror request back

session management

http is stateless
use cookies (sent by server; replied by client)
use query strings (but not stored on client like cookies)
must protect tokens using SSl
else eavesdropping, replying, manipulating possible

web application framework
separation of concerns (layout, style, content)
architecture (logging, caching, separation of concerns)
security (firewalls, CSRF, RBAC, validation, secret management)
scaffolding (code generation)

8.5.2 vulerabilities

unvalidated input
broken access control
broken authentication / session management
cross site scriping (XSS)
buffer overflows
insufficient transport layer protection
injection flaws
insecure direct object reference
failure to restrict URL access
improper error handling
insecure cryptographic storage
denial of service
insecure configuration management

8.5.3 attacks

XSS
get web site to display user content from different origin
script embedded on page based on unvalidated input
examples are calling URL of attacker with secret as argument
persistent attacks display stored value (user post)
reflected attack display input value (link with displayed parameters)

CSRF
user is logged in server s1
user accesses server s2 wich refers to ressource on server s1
if ressource has been accessed, CSRF declared successful

8.5.4 input

user input sent to web server & passed to backend

prevent injection
treat all input as malicious
canonicalization (normalize, like ./foo = foo)
whitelists, max/min
separate control & data channel

8.5.5 authentication

user authenticated against server

basic authentication
server checks access through header
header contains base64 encoded cleartext credentials
but no time-out, logout

form-based authentication
web application checks access
but not standardized & flawed implementations

prevent brute-force
captchas but ML / crowdsourced solving
timeouts but because can deny access

prevent intersection
secure connection (SSL)

8.5.6 session handling

token with state or linked by identifier to server state

prevent highjack (others session used)
secure transport with SSL

prevent prediction (others session id predicted)
use high entropy encodings

prevent fixation (others authenticate prepared session)
create session only after authentication

8.5.7 information flow

flow of data between multiple applications

prevent CSRF (user)
enforce match of domain, protocol & port (same origin policy)

prevent CSRF (developer)
use csrf token in form

prevent XSS (user)
disable scripting (but reduced functionality)
no promiscuous surfing (know who you can trust)

prevent XSS (developer)
CSP (define whitelist of trusted content sources)
HttpOnly flag to deny javascript access to cookies
sanitize & escape inputs/output

9 risk management

protect organisation and its ability to perform its mission
carried out at different levels, granularities and scopes
interative activity, input/output of all phases of development

9.1 target

maintain customer, stockholder, taxpayer confidence
protect CIA of sensitive data
avoid thirdparty liability by illegal/malicious acts
avoid misuse, disruption of network, computer, data
comply with laws and regulations

9.2 approach

identify risks to assets
allocate countermeasures based on cost & impact to risk

9.3 balance

functional requirements & usability vs costs & risks
theoretical (cryptoanalysis) vs practical (dictionary attacks) risks
high (widespread attack) vs low (single user compromised) impact

risk handling options
avoid (safegards, clean design)
transfer (allocate it to other parties)
assume (accept it & create contingency plan)

9.4 elements

assets (db)
elements which need to be protected
tanglible (products, buildings, systems (hard & software))
intanglible (information, people, reputation, trust, political stance)

vulnerability (no input)
aspect of asset which is exploited by threat

attacks
exploits vulnerabilities of assets

threat (harmful input)
cause unwanted event to assets
cause attacks using vulnerabilities
may be intentional (hacking) or not (market crash)
a "danger" if exploitable vulnerability & dangerous
a "reduced risk" if countermeasure against vulnerability

threat agents (hacker)
sources of threats
humans (hacking, negligence)
nature (flood)
environment (equipment failure)

countermeasures (prepared statement)
detect, deter, deny attacks (also attackable)
direct cost (auditing, intrusion detection)
impact behaviour (encryption, authentication)

owners (bank)
have assets & coordinate countermeasures to attacks

9.5 risk

expected loss to asset from exploited vulnerability
risk = likelihood $*$damage of event

asset valuation
what an asset is worth to self & competitors
cost of production, loss, modification, publication
value now/tomorrow (operations, price in open market)

intanglible value (reputation, political, trust)

handling
avoid (reduce value, increase defenses, stop operation)
transfer (insurance, other organisational units)
accept (mitigate/reduce it, or simply accept)

annual loss expectancy (ALE)
multiply damange with expected annual occurrence
ignores dependency of events, distribution, variance

enablers
design/implementation flaws
misconfiguration / mismanagement
inadequate policies/enforcement
lack of protection & training

9.6 risk analysis

examine a system, its operational context
determine possible exposures & potential harm

9.6.1 procedure

identify assets
ascertain threats and corresponding vulnerabilities
calculate / priorize the risk and how to handle it
for assumed risk implement countermeasures
monitor the effectiveness

9.6.2 quantitative analysis

assign independently obtained, objective, nummeric values
define asset value, potential loss, safeguard cost, probability
enables cost/benefit analysis, communicates values & risks
but costly, inaccurate (intanglible, false sense of security)

9.6.3 qualitative analysis

rank potential threats within categories of likelyness (low - high)
simpler because do not need exact value & probabilities
thinking about problem more important than explicit numbers
but more subjective, cost/benefit analysis not possible

asset valuation (get relevant assets)
to get impact if assets were compromised
for each business operation (legal, financial, ...)
then for each CIA property (disclosure, modification, ...)
define valuation (2k loss = 3, 5k loss = 4, ...)
summarize for each CIA property the resulting mean

risk evaluation (get relevant threats)
identify importance threshhold and focus on critical
determine the impact (based on probability of threats to assets)
consider with or without existing countermeasures

risk mangement
consider countermeasures with cost/benefit

9.7 tools

9.7.1 risk factor

for each potential outcome, calculate cost
multiply cost with probaility of outcome
sum up all values to get risk factor

9.7.2 OCTAVE

create tree like asset → access → actor → motive → outcome → impact
then create a risk mitigation plan for each property
does not consider probabilities

9.7.3 OWSAP risk rating

attacker factors
skill
motive
opportunity (ressources needed)
size (potential attackers)

vulnerability factors
ease of discovery
ease of exploit
awareness
intrusion detection

9.7.4 vulnerability score

combine probability & impact; each low - high; in same table
define value for each table cell

9.7.5 NIST categorization

likelihood
high / medium if threat actor motivated
low if remedies effective

magnitude
high if execution leads to death or drastic injury
medium if execution is costly, violates mission
low if exercise affects mission

10 safety engineering

10.1 definitions

faillure
deviation from specification

fault
reason for faillure

faillure mode
the way element fails; breakage, leak

10.2 risk priority number (RPN)

occurrence (relative probability)
severity (impact of wort possible outcome)
detection (probability that failure is visible)
scale aspects from 1-10
criticality = occurrence ∗severity
RPN = criticality ∗non-detectability

10.3 failure modes and effect analysis (FMEA)

bottom-up, textual approach
to identify root causes of faults early
used in mechanical / hardware oriented systems

process
decompose recursively into analysable subcomponents
analyse their individual faults
check their effects on the system (potentially with RPN)

example
decompose clock recursively into components & subcomponents
for each screw, analyse how it could break
check what impact that would have on showing time

10.4 fault tree analysis (FTA)

deductive top-down, qualitative and quantitative
to identify sources of system failure
done at design / architecture time

process
determine top-level failures, malfunctions
find intermediate events leading to next upper with AND, OR gates
recursively construct tree until at primary events

primary events (leaves)
basic events (probabilistic occurrence; like bitflips)
undeveloped events (events without major effect; like lamp fails)
external events (expected events; like weather event)

tree
many equivalent trees possible (due to boolean algebra)
use tree that facilitates discussion the most
expand complicated events in their own trees

sets
cut set (events which together lead to system failure)
minimal cut (cut set where no event can be removed anymore)
path set (events which together prevent system failure)

identify cut sets
identify
called minimal cut iff no longer a cut if element is removed
probability AND = P1 ∗P2
probability OR = P1 + P2 - P1 ∗P2

path sets
events that if none of them occurrs, no system failure
invert boolean formula to identify them

example
top node = gain priviledged access

subnodes = OR (social engineering, physical access)
recursively refine nodes

10.5 threat modeling

identify possible attacks (sniffing, brute-force)
to assets (customer data, credentials)
using channels (webshop, database)
need to find hot spots because combinatorial explosion

approach
consult deployment diagrams, system documentation, source code
do interviews
find out which systems (shop, db)
use which channels (http, https)
at various abstraction levels (applications, network)
identify data pathways to determine CIA for channels

benefits
make critical components transparent to all stakeholders
do comprehensive review instead of selective discussions

UML
use <<system>>stereotypes on nodes, <<pathway>>on associations
CIA as tagged values

10.6 attack modeling

vulnerability catalogs / checklists
be creative, ask specialists
assume an adversary

resources
BSI IT-Grundschutz-Kataloge / Gefährdungskataloge
OWASP Top Ten Project
Common Weakness Enumeration from mitre

tables
columns include target, attacker, attack, impact and countermeasures
well suited to communicate threats, but less helpful to identify new ones

trees
like fault trees plus roles/motives of attacker
top-level nodes obtained from misuse cases
each node is an attack with probability ∗impact = risk
refine to composite (AND, <<include>) attacks; min p, max i
refine to alternative (OR, arrows) attacks; max p, max i
makes critical points of the application clear

10.7 tools

data pathways
use elements from deployment diagram
annotate communication paths with <<pathway>>
define CIA for <<system>>, <<pathway>>
trust boundaries where integrity value changes

10.8 MIL-STD-1629A

military FMEA procedure
1. define analysed system (mission, interfaces, performance, constrains, ...)
2. construct block diagram of systems
3. identify item / interface definitions
4. evaluate and rank failures
5. identify causes and actions
6. take actions to reduce RPN
7. document

evalute failures & identify causes
identify possible failure modes
identify effects of them (for severity)
identify causes of them (for occurrence)
evalute control process (for detectability)
determine RPN
identify actions leading to improvement

failure categories
catastrophic (death or system loss)
critical (severe damage or mission loss)
marginal (minor damage or mission degradation)
minor (no damage but maintenance/repair required)

example car headlight
failure mode = light off
effects = inoperable at night (8)
causes = battery dead (8), broken wire (3), ...
controls = noticable during night (6)
RPN = 8∗8∗6 for battery dead, 8∗3∗6 for broken wire ...
improve detectability with indicator in console

11 security design

ensure effectiveness of controls / safeguards
ensure problems detected early
ensure recovery functional

11.1 safegards

authentication, authorization (disk encryption)
system security architecture (firewalls, VPN)
secure communication (encryption)
physical security (guards)
interruption prevention (backup generators)
procedural measures (training)
secure coding (code scanners, static analysis)

11.2 process

assure
do security reviews, static analysis
do (penetration) testing (like NIST 800-42)

detect
engage intrusion detection systems & virus scanners
perform external audits

recover
restore operations quickly using prepared processes

11.3 principles

creative process; use/hire/read experience

11.3.1 best practices

keep it simple (allows easier overview)
deny by default (if no explicit confirmation, deny)
least privilege (no root if not needed)
define baselines & enforce them (like encryption standard)
open design (many can convince themselves)
separation of priviledge (four-eyes principle)
least common mechanism (avoid shared state)
accessability (easy-to use APIs to prevent errors)
defense in depth (at all layers of system)
use standard & mature tools (open source libraries)
generate security code (if framework convoluted)
validate all input (direct, database, config, network, registry input)

collections
BSI IT-Grundschutz-Kataloge / Gefährdungskataloge
Open Web Application Security Project (OWASP)

11.3.2 implementation options

use standards
integrate standard security mechnisms (like use HTTPS)
can defend against generic threats

roll your own
implement security in application
best for IO validation, access control, encryption
fits perfectly to application, simplifies architecture
but expensive to develop, test; error-prone

refactor
simplify design & differentiate between criticalities
but expensive to develop & use

11.4 example

11.4.1 mars rover

explicit code style & enforcement
certification (concerning code rules)
CI (coverity, codeSonar, semmle & uno)
code reviews (tool supported & manual)
tests (unit, integration, ...)
model checking (spin automatic/manual construction)

tools
language compliance (no compiler extensions, no warnings)

predictable execution (statically verifiable upper bounds)
defensive coding (assertion desity; remained in prod)
restrict language (no function pointers, preprocesser, indirection)
MISRA C coding guidelines & static analysis

statistics
84% of detected errors lead to code changes
12% explicit disagreement

11.4.2 further ressources

Chess/West book
Hovemeyer, Pugh, Finding Bugs is Easy

12 code scanning

pragmatic static analysis (no runtime analysis)

12.1 definitions

failures (deviation of behaviour detectable at system interface)
errors (deviation of systems behaviour from intended one)

12.2 scope

type checks
check assign works type-wise
but false positives/negatives; undecidable

style checks
check that switch-case with enumerations complete
more picker, superficial rules

program understanding
analyse code & history of commits
find hotspots of programs

property checking
check behaviours against property specification
like memory cleaned up before returning
but state explosion, complex to create models

bug patterns
null pointer dereference
return values not considered

12.3 defensive programming

check all arguments for valid content (null & format)
code scanners can support the tedious checking of all cases

12.4 static analysis

generic defects (buffer overflows)
context-specific (mishandling credentials)

problem categories
input validation / representation (overflows, XSS)
API abuse (disrepected contract concerning in/output type)
security features (hardcoded passwords)
time and state (race conditions)
error handling (missing, poorly)
code quality (infinite loops, null pointers)
encapsulation (lack of it)

hardness
behaviour in general undecidable
hence need over/underapproximation to terminate

targets in practice
find errors, not prove absence (unsoundness OK)
usability & simplicity

challenges in practice
find & compile code (proprietary tools, weird build processes)
parsing code (many compilers & versions)
understanding semantics (detect external libraries, syscalls)
provide usability (user must understand & care about bug)
guarantee low tool output changes (many changes frustrating)
keep false positives low (less than 30%)

tooling usage
set goals from a risk analysis perspective
use output of tools in code reviews to fix it
augment with product specific rules
place burden of proof for false positive on committer

12.5 representations

token string
get stream of identifers
can detect dangerous functions

parse tree
get tree from identifiers

abstract syntax tree
normalizes language; simplifications
at the same time create symbol table
can do type checking (like compilers, linters)
can use structural rules (field=static & type=connection)

control flow graph
basic blocks (uninterrupted sequence of instructions)
forward edges (connection between basic blocks)
backward edges (possible loops)
get all possible ways of execution (called trace)

function call graph
functions as nodes, invocations as edges
use dataflow analysis to find out traces

12.6 dataflow

determine how data moves to derive properties

12.6.1 desired properties

annotate with assertions, rules
infer from list of common bugs (fault models)
infer based on consistency

12.6.2 techniques

static single assignment
introduce new variable for each assignment
to see where data comes from

constant propagation
replace constant variables with their value
to detect hardcoded strings; like credentials

branching
selection function (a = $(a1, a2)) at branch join
then continue analysis with one of these values
to check all potential dataflow executions

symbolic execution
determine weakest precondition
such that result has desired property
check variable assignments, branching and function calls
but #paths is exponential in #conditions

12.6.3 properties to track

range
check in which range variable can be

pointer aliasing
determine which pointers point to same data location

type state
check variable for state (undefined, allocated, ...)

taints
variable properties as flags (taints) like unsafe, \0 terminated
rules add/subtract taint flags
source rules (where tainted data enters system)
sink rule (where tainted data would be problematic)
pass-through rules (how functions manipulate tainted data)
cleanse rules (functions which remove taint)
postcondition cin (source) = value tainted
postcondition strcpy (pass-through rule) = if input tainted then result tainted
precondition printf (sink) = value must not be tainted

13 testing

detect flaws by trying to refute some hypothesis

13.1 flaws

design
formal methods (model checking, theorem proving)

implementation

static analysis (inspect code) like compilers, code scanners
dynamic analysis (execute code) like testing, run-time analysis

13.2 testing types

black box testing (programs as input/output mapping)
white box testing (source code known)

13.3 testing is finite

because chosen input & runtime must be finite

infinite breadth
infinitely many executions
"some execution commits transaction"
cannot refute existential (because maybe just not found yet)
cannot verify universal (because did not test all yet)

infinite depth (liveness)
infinitely long execution
"execution eventually terminates"
if terminates, then not refuted
if no termination, could terminate later hence not refuted
need safety property (bounded time & repetitions)
can not verify liveness ("eventually")

13.4 test generation

systematic approach to test selection

13.4.1 method

random
fault-based with a fault model for P
model-based with a formal model for P
specification-based with a formal model for P

13.4.2 fault model

describes class of faults
each fault model justifies small number of mistakes
choose fault models that could impact the application

receipe
identify input domain
choose fault model for input domain
partition input domain using fault model(s)
select representative test for each partition
compare with test oracle (given by specification)

examples
check boundary inputs values (pick max, min, turning points)
check output values (pick value for each output partition)

security example
broken authentication & session management
"session ID in URL" → check if url can be used for authentication
"session ID no timeout" → check if session can be resumed later

ressources
attack surface analyser from microsoft

advantages
document & learn from the past (& from others)
make failures visible by looking for faults

13.4.3 adequacy criteria

select finite set S from infinite input domain D
want to know if S is adequate set of tests
ideally, S should expose all faults

13.4.3.1 criterion

must be measurable, reliable and predictive
applications include measuring testing progress / quality

13.4.3.2 flawed test criterion

no budget / time left
testers / user find no issues

13.4.3.3 coverage

is not predictive (uncovered parts are not detected)

statement
% of statements hit
variations include branch, path, DV pairs
could still miss relevant input value (division by 0)

specification
% of specification exercised
so a test must exist for all allowed / disallowed specs

model
% of model component exercised
variations include state, transition, loop-free paths

13.4.3.4 mutation analysis

mutate program P to get P1 (for example, replace + → -)
some mutants e could be semantically equivalent
mutants should produce different output for t ("detected")
adequancy of test t set = mutants_detected / n

assumes
competent programmer (which makes syntactical mistakes)
coupling effect (syntactical mistakes produce faults)

14 security testing

refute requirement in presence of adversary

14.1 terminology

requirements (what) motivate specification (how) under assumptions
specification must be implemented correctly (S-Test)
assumptions have to hold (E-Test)

example
requirement is enter → authorized
specification is door_open → has_card
assumption1 is enter → door_open
assumption2 has_card → authorized

14.2 S-Test

refutes that system meets specification
independent of adversary model
measure adequacy with coverage, mutation analysis
fix implementation when issues are found

14.2.1 fault-based

refute specific detail or general "no crashes"
usually not able to test guidelines because too general
corner cases often rely on unexpected/unspecified assumptions

14.2.2 risk-based

obtaining spec by yourself and then test if system fulfills

process
gather risks and countermeasures from risk analysis
derive requirements and specifications, respectively
create test plan checking specification, priorised by risk
can not check original env assumptions and adversary model

example
router has risk of unauthorized access
hence requirement that only authorized users can interact
countermeasure (hence specification) include strong password
testing needs to check if that is the case

14.2.3 fault-injection

break systems dependences
inject faults with depedencies and observe program

examples
security dll of IE removed has no consequences
if some decision part fails, must fault whole decision

14.2.4 fuzz-testing

break system using malformed input
use wrong size, type, order of inputs
combine with code coverage tool to generate inputs for all paths

requirements
fault/vulnerability model for specific system
which failures can be checked generally (like crash or not)

approach
mutate valid input using a fault model
ensure mutation not readily discarded by parsers
observe generic failures based on memory access (not input / output)

example
collect PDFs online (valid inputs)

mutate some bits / stream (fault model)
check that it does not crash (generic failure)

14.2.5 vulnerability-driven tests

check specific vulnerabilities of target system
can automate using nessus, nmap, burf, w3af, kali linux
but need few false positives else trust is lost

14.3 E-Test

refute that assumptions hold (which are often unclear)
depends on environment, system and adversary model
adequacy determined by adequacy of assumptions
fix design when issue is found

environments
do not admit delimitation
hence can not formally describe what adversary can do
can account only for limited number of interactions
hence rely on closed-world assumption (CWA)
refuting CWA depends on testers capabilities (creative work)

side-channel
unanticipated communication channel, such as timing
depends on capability of adversary
include it in I/O specification to convert to nominal channel
for example formatting server, bribing personnel

challenges
assumptions hard to explicate
unclear how well CWA is tested

15 evaluation criteria

15.1 types of assurance (low to high)

device was build by trained people
process used is sound, and results in good product
testing of device confirms properties

15.2 standards

for products, services and processes
assure that system has specific properties

gain confidence
indirectly by observing processes
directly by evaluating product

15.3 NIST

guidelines & principles for secure systems

800-14
specifies principles & practices
like "cost-effectiveness" & "do risk analysis"
useful starting point

800-123
guide to general server security
like "remove unnecessary services"
technical description to system hardening

15.4 ISO/IEC 27000 series

code of practice for information systems (certification possible)
high-level & management-oriented
security as a process, continuous improvement
need top-management support to avoid mismatched controls
does not check products; assumes these follow from processes

27001 (requirements)
plan (fix scope, security policy, risk analysis method)
do (do risk assessment & improvements)
check (compare results against objectives; audits)
act (improve system)

audit process
preliminary, informal review
detailed compliance audit with list of problems or certificate
follow-up reviews to ensure compliance help up

27002 (code of practice)
to establish security policy
asset classification & control
personell, environmental security
communication/operations management
access control
system development and maintenance
for example change control for access control

15.5 common criteria

evaluate products / systems with security functionality
specify clear actions during development cycle
ensures comparable results over all reviews (peer-reviewed)

15.6 example

ISO 9001 (quality management system)
best practice catalog for which expert testifies best effort
identify customer requirements
formulate quality objectives
controlling purchasing processes
support internal communication

eu directives
personal data protection & GDPR
electronic signature
money laundering

basel committee
risk management principles
basel II
outsourcing of financial services

security standards
BSI IT Baseline (develop & evaluate process & systems)

16 appendix

16.1 actionGUI

generate java application from three different layers

data model

```
entity Message {
String login
Role role
Set(Message) messages oppositeTo messageOwner
}
```

security model

```
role USER {
Message {
create
read(title), read(text) constrainedBy [self.owner = caller and !(value
add(messageReplies) constrainedBy [self.owner = target.owner and calle
}
}
```

GUI model

```
Button RegisterMod_B {
String text := ['Register as Moderator']
event onClick {
if [(not $Name_TF.text$.oclIsUndefined())] {
if[Person.allInstances()->forAll(c|c.login <> $Name_TF.text$)] {
newUser := new Person
[$newUser$.login] := [$Name_TF.text$]
[$newUser$.roles] += [Role::ADMIN]
open MainWindow(caller:[$newUser$], role:[$newUser$.personalRole])
} else {
notification(['Error'],['Username already exists.'],[500])
Name_TF.text := [null]
Password_TF.text := [null]
}
} else {
notification(['Error'],['The form contains errors.'],[500])
}
}
}

Table Table_T {
Set(Category) rows := [Category.allInstances()]
Category selected  := [null]
```

```
columns {
['Name'] : Label name {
event onView (text){
try{
text := [$Table_T.row$.name]
} catch (SecurityException){
text := ['RESTRICTED']
}
}
}
}
}
```