Auditing Software

Ibéria Medeiros

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

Motivation

- “All software projects are guaranteed to have one artifact in common – source code. Together with architectural risk analysis, ***code review for security*** ranks very high on the list of software security best practices.”
  - Brian Chess & Gary McGraw

- “during the Windows Security Push (…) [Feb-Mar 2002] we found that the most important aspect of the software design process, from a security viewpoint, is ***threat [attack] modeling***”
  - Howard & LeBlanc

(Manual) Software Auditing or Code Review

- Aims at identifying security flaws along with its root causes in software projects at the
  - ☞ design phase (also called design review)
  - ☞ during/after the implementation (also called application review)

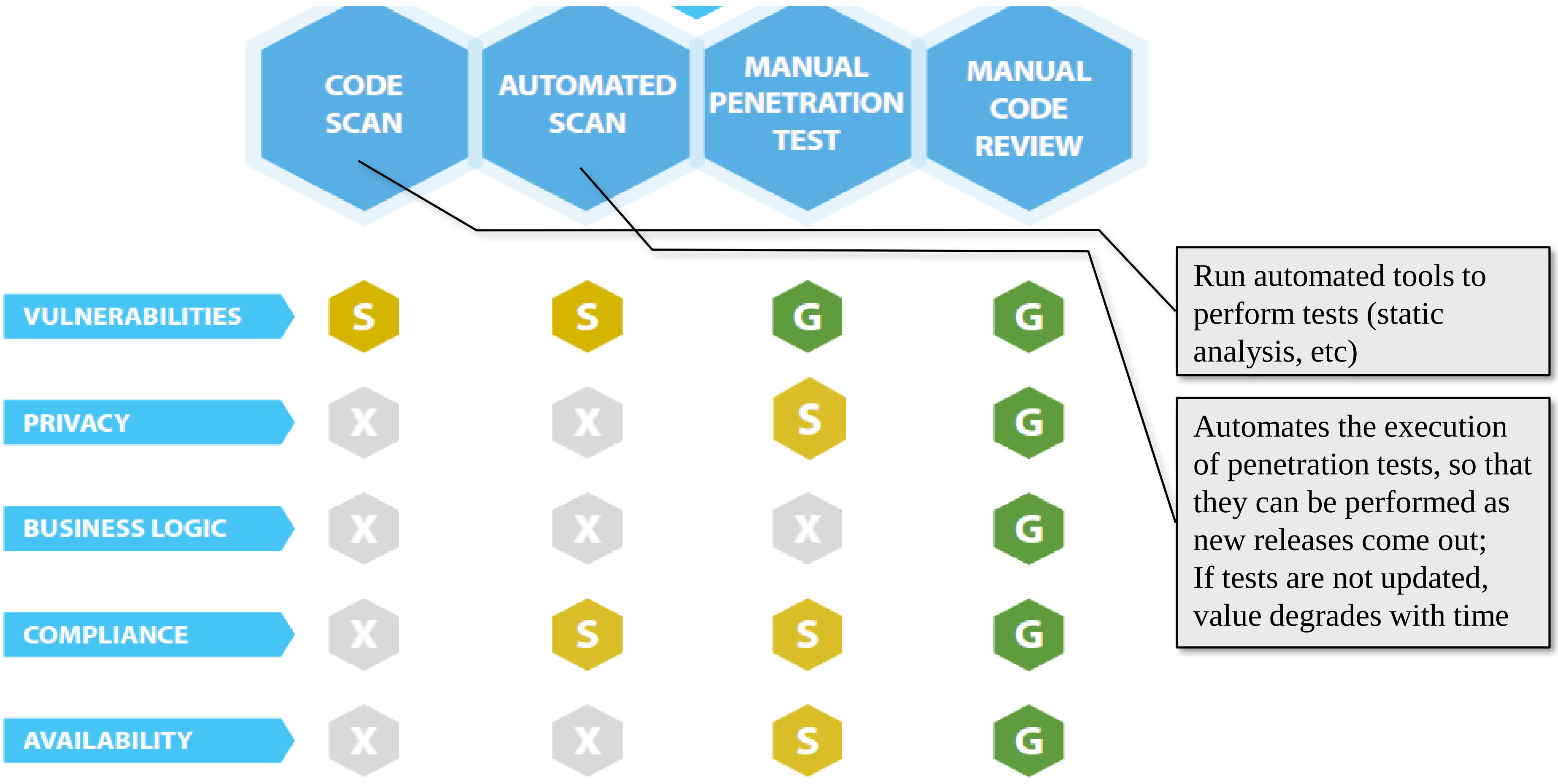
- Helpful to
  1. determine if the **proper security and logical controls are present**, work as expected and they are invoked in the right places
  2. allows **more bugs to be catch early** in the software development life cycle (SDLC)
  3. assures software programmers that an organization has decided to **follow secure development techniques**
  4. can give an insight into the **"real risk"** associated with the code

Code Review

- The **context of the software** must be provided, allowing a tester to understand what is being assessed
- Becomes possible to give a **real estimate of the likelihood** of the (successful) attack and the impact of the breach
- Remediation can be performed based on the priority of the identified issues (potentially avoiding to fix everything)
- It is a white box testing technique that **should utilize technology**, such as tools for static analysis, but the tester needs to verify every result to ensure that there is a real issue

> *"Over the last 10 years, the team involved with the OWASP Code Review Project has performed thousands of application reviews, and found that every non-trivial application has had security vulnerabilities."* OWASP Code Review, 2017

We Can´t HacK Ourselves Secure



	CODE SCAN	AUTOMATED SCAN	MANUAL PENETRATION TEST	MANUAL CODE REVIEW
VULNERABILITIES	S	S	G	G
PRIVACY	X	X	S	G
BUSINESS LOGIC	X	X	X	G
COMPLIANCE	X	S	S	G
AVAILABILITY	X	X	S	G

Run automated tools to perform tests (static analysis, etc)

Automates the execution of penetration tests, so that they can be performed as new releases come out; If tests are not updated, value degrades with time

S *Some chance of finding the problem*

G *Good chance of finding the problem*

(Manual) Software Auditing

- In the rest of this lesson we will mainly focus
  1. threat/attack modeling approach
  2. strategies for manual review of application code

THREAD/ATTACK MODELING

Threat/Attack modeling

- **Objective** is to model threats, i.e., to *identify and characterize how attacks can affect the system*
  - ☞ abstract away lots of details to get the bigger picture
  - ☞ allows the discovery of issues before the system is built
  - ☞ lets you anticipate the threats that may affect the system

- Relevant questions
  - ☞ What are you building?
  - ☞ What can go wrong?
  - ☞ What should you do about those things that can go wrong?
  - ☞ Did you do a decent job of analysis?

- Attack modeling serves as basis for **risk management**
- Should be done early in the software development cycle (recommended) or when the system is about to be rolled out

Attack modeling (cont.)

- Why do we need attack modeling?

  If we can understand all different ways in which a system can be attacked, we can address those attacks!

  This can be achieved by either

  ☞ ***mitigate the attack:*** *placing the protection/control mechanisms to prevent the exploitation*

  ☞ ***eliminate attack:*** *removing the vulnerability (e.g., by eliminating a feature of the system)*

  ☞ ***transferring the attack:*** *let someone or something else handle the risk (e.g., use other system to do authentication)*

  ☞ ***accept the risk:*** *accept that for some attacks you are willing to live with the risk*

Attack/Threat Trees

- Fault trees are used in dependability and software safety to identify failure modes

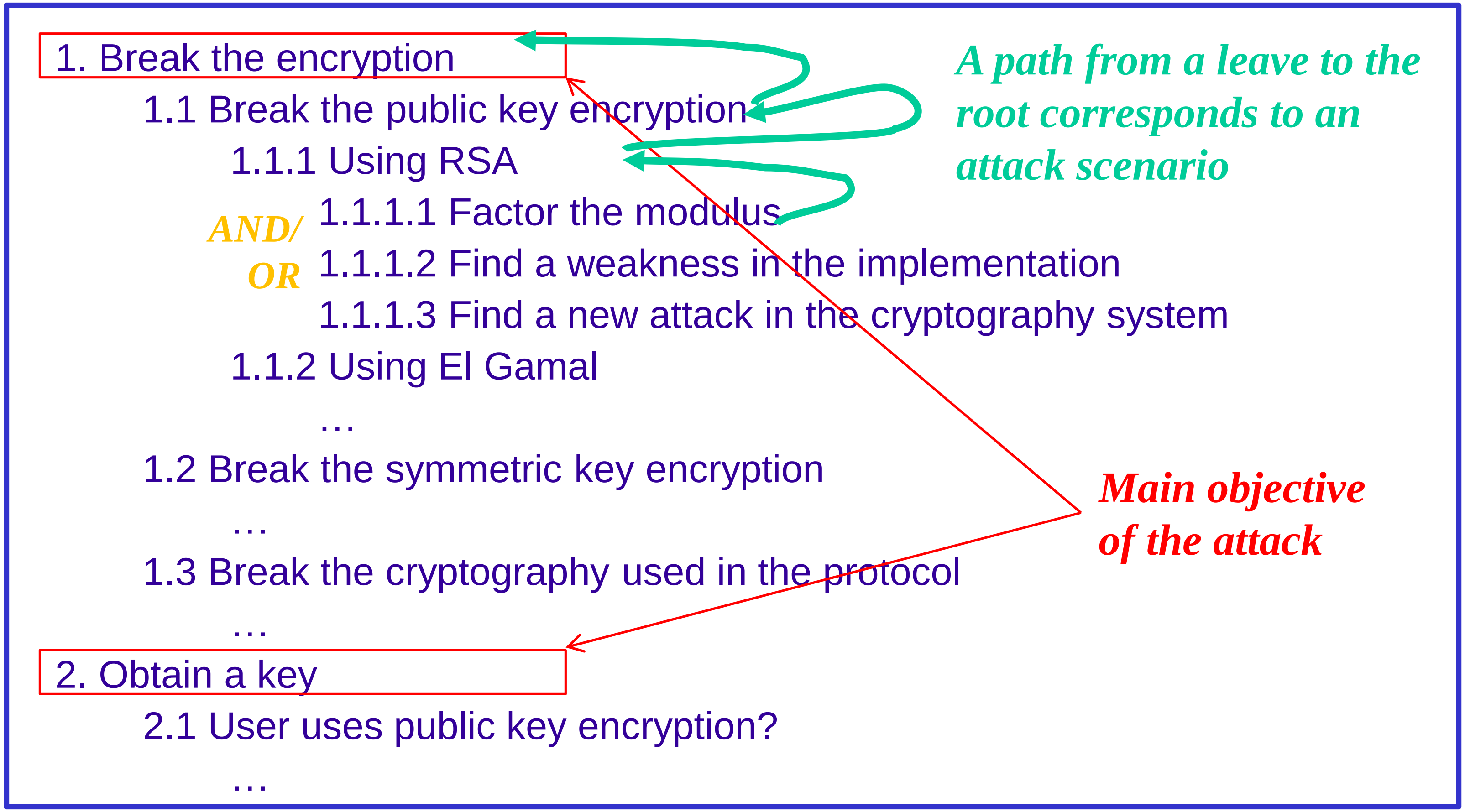
- B. Schneier proposed concept of attack trees
  - ☞ “Attack trees”, Dr. Dobb’s Journal, Dec. 1999

- Threat trees / threat modeling
  - ☞ the same as attack trees/modeling
  - ☞ the terms were introduced by E. Amoroso earlier (1994)
  - ☞ these terms are much more used than attack trees/modeling since they are pushed by Microsoft

- Best term is probably attack, not threat

Schneier-style attack tree

- The tree identifies paths of attack to attain a certain goal (there is also a graphical representation)



1. Break the encryption

    1.1 Break the public key encryption

        1.1.1 Using RSA

            1.1.1.1 Factor the modulus

            1.1.1.2 Find a weakness in the implementation

            1.1.1.3 Find a new attack in the cryptography system

        1.1.2 Using El Gamal

            …

    1.2 Break the symmetric key encryption

        …

    1.3 Break the cryptography used in the protocol

        …

2. Obtain a key

    2.1 User uses public key encryption?

        …

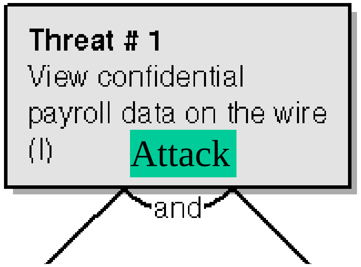
AND/ OR

A path from a leave to the root corresponds to an attack scenario

Main objective of the attack

Microsoft-style threat tree

- The tree contains both the **vulnerabilities exploited** and the **attack steps**



Threat # 1
View confidential payroll data on the wire (I)

Attack

and

Equivalent to Schneier's, but this is the one we're going to consider

Steps for the Analysis

- ► Step 0: define the scope and information gathering
- ► Step 1: decompose the application into some useful representation
- ► Step 2: identify vulnerabilities (or attack objectives)
- ► Step 3: build attack tree(s) for each target
- ► Step 4: order vulnerabilities in terms of risk

- It is a long and tedious process! But …

Step 0: Information gathering

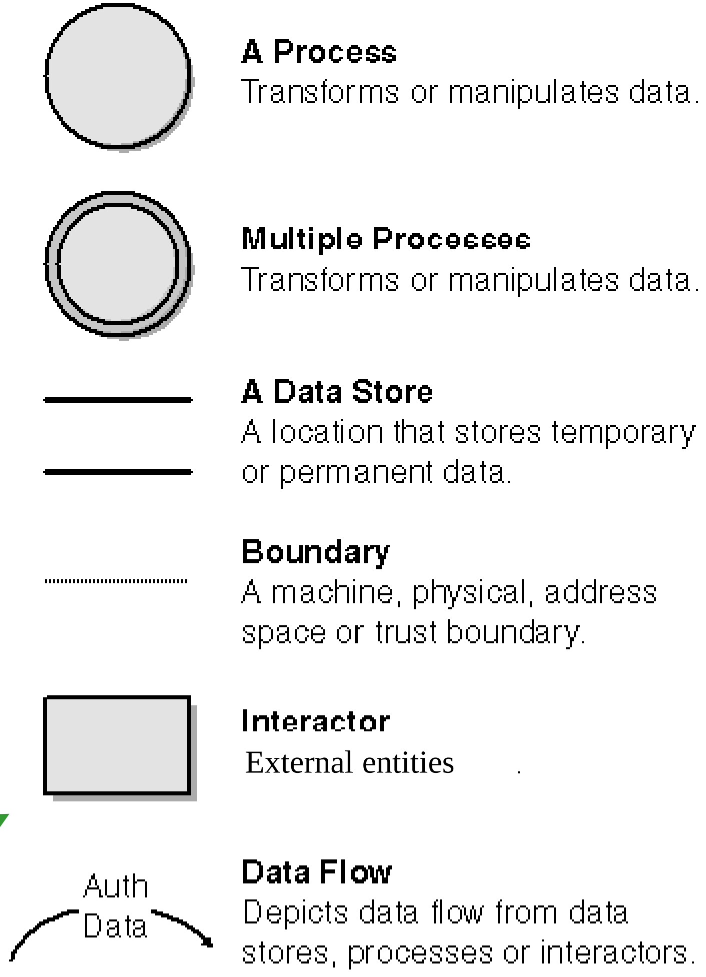
- Compile information about the application
  - assets: things that might be valued by an attacker – data (e.g., credit cards), components that allow the execution of code
  - entry points: ports, RPC endpoints, submitted files
  - external entities: user classes and external systems that interact with the application
  - external trust levels: privileges granted to external entities
  - major components
  - use scenarios
- Sources of information
  - interviews with the developers, developers documentation, design documents, source profiling, system profiling

Step 1: Decompose application

- There is no unique solution to represent the decomposed application, but one possibility is Data Flow Diagrams (DFDs) or UML

- Decompose system in components that make sense from an architectural point of view, components in subcomponents …
  - ☞ not too many levels!

- main symbols



A Process
Transforms or manipulates data.

Multiple Processes
Transforms or manipulates data.

A Data Store
A location that stores temporary or permanent data.

Boundary
A machine, physical, address space or trust boundary.

Interactor
External entities .

Data Flow
Depicts data flow from data stores, processes or interactors.

Example app. decomposition (level 0)

Case study: payroll application

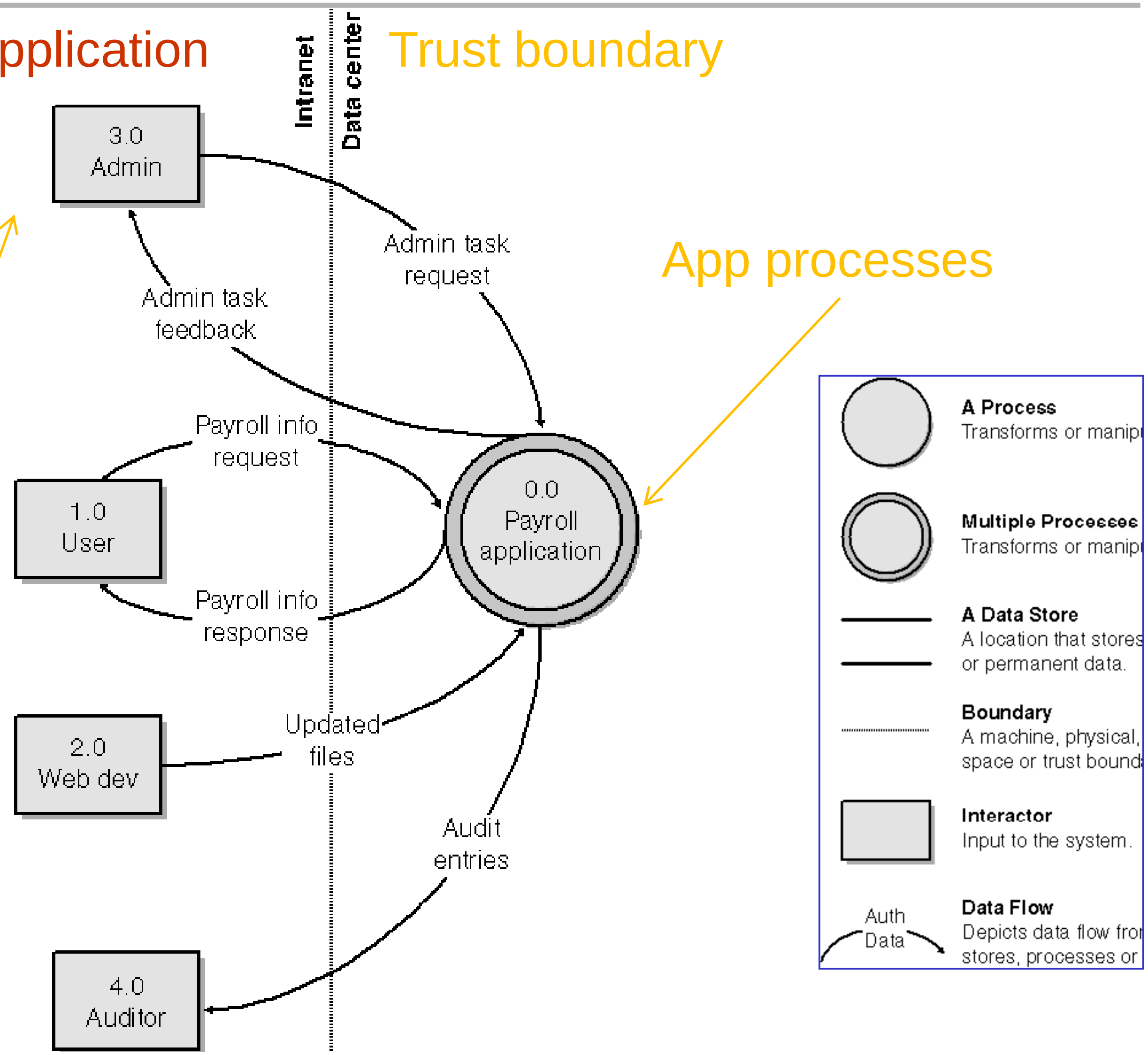
Level 0 DFD

External entities

Number each identity to make it simple to identify later on

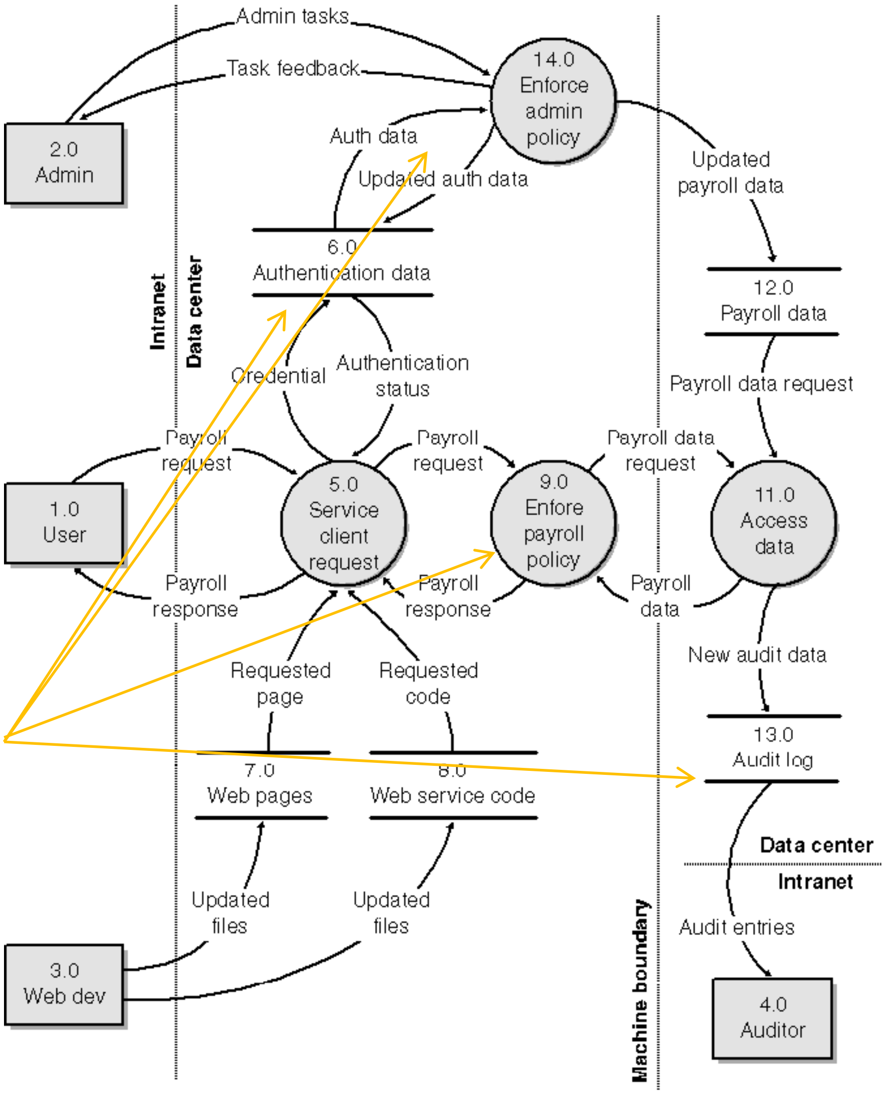
Trust boundary

App processes



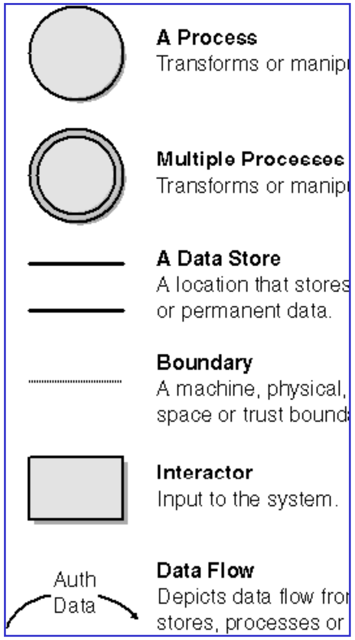
Example app. decomposition (level 1)

Level 1 DFD



Initially you don't know anything about the app. What does the diagram tell you? Can you find security related components?

Very relevant components for security operations



Step 2: Identify vulnerabilities

- Identify potential vulnerabilities for each interaction and component of the decomposition of step 1

- This is the most problematic step since we need to brainstorm to find the vulnerabilities
  - ☞ how do we remember all of them?
  - ☞ what about novel types of vulnerabilities?
- A good approach is to use a vulnerability taxonomy
  - ☞ there are many
  - ☞ e.g., CWE – Common Weakness Enumeration
  - ☞ e.g., Microsoft’s STRIDE – classifies vulnerabilities in terms of their effect

STRIDE taxonomy

- **S**poofing identity
  - allows an attacker to pose as a valid entity
- **T**ampering with data
  - malicious modification of data
- **R**epudiation
  - possibility of denying to produce a certain event
- **I**nformation disclosure
  - exposure of information to entities that were not supposed to
- **D**enial of service
  - negation of some component’s service to users/components
- **E**levation of privilege
  - increase of the privileges of the attacker

Tips to Stay on Track

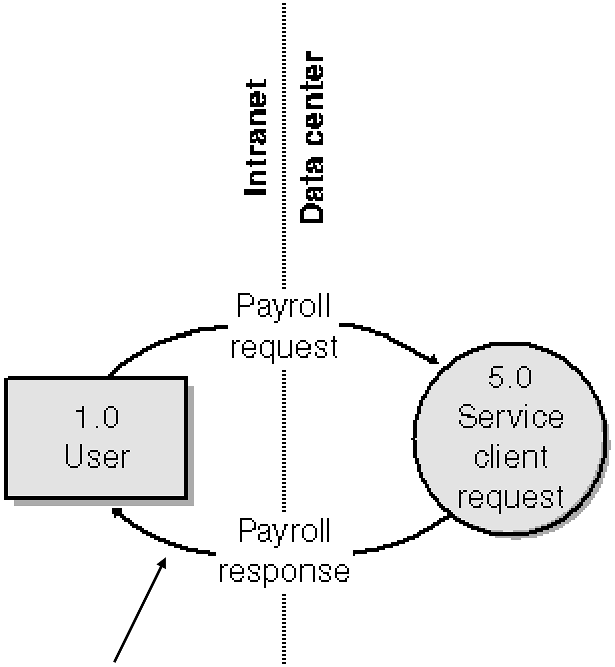
- Keep some order when performing the analysis
  - ☞ which entities should you analyze first? Maybe, the external?
  - ☞ follow the threats (S->T-> …) in order

- Never ignore a threat
  - ☞ even if you are not looking at that threat at this point
  - ☞ write it down and return later on

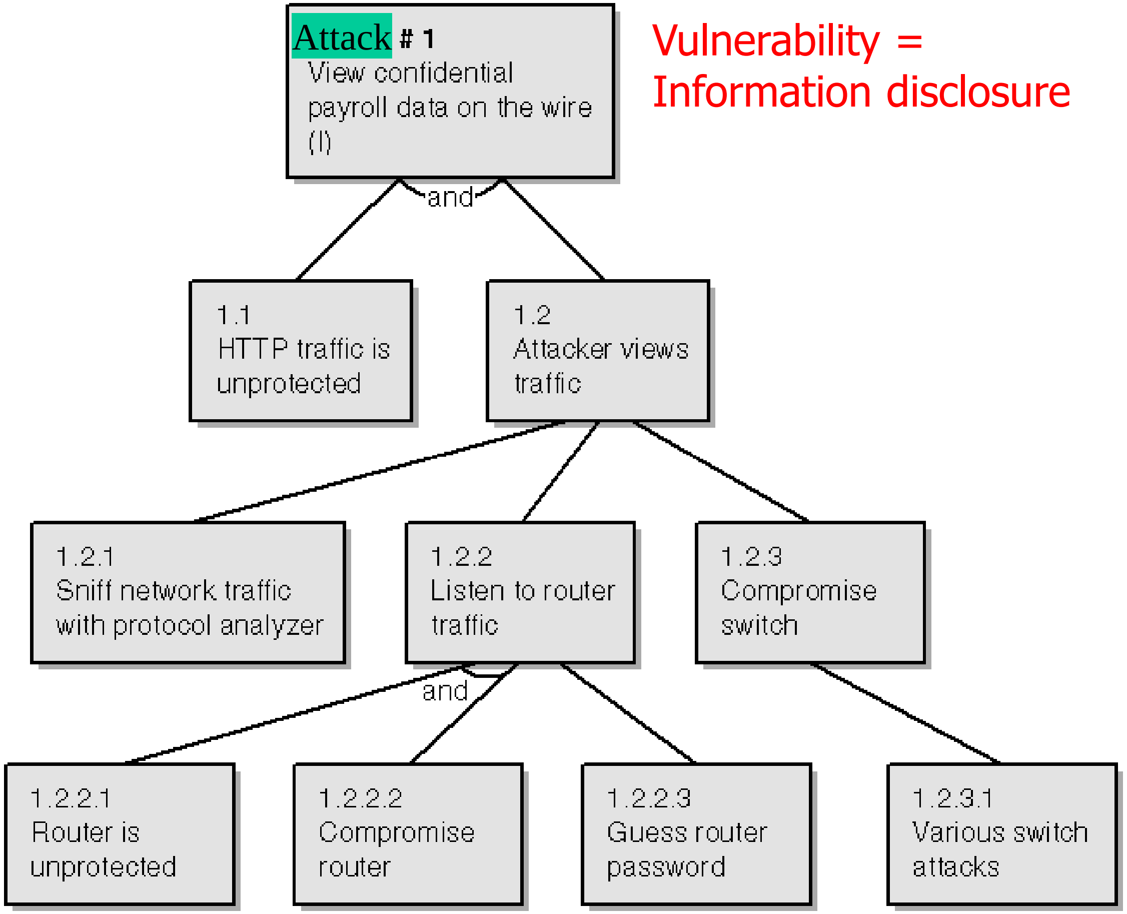
- Focus on feasible threats
  - ☞ give priority to threats that are really relevant and that you can do something about
  - ☞ e.g., *should you worry if someone modified the CPU chip?*

Step 3: Build attack trees

- For each identified vulnerability, one needs to build a corresponding attack tree that explains how it could be exploited



Excerpt of level 1 decomposition (an interaction)

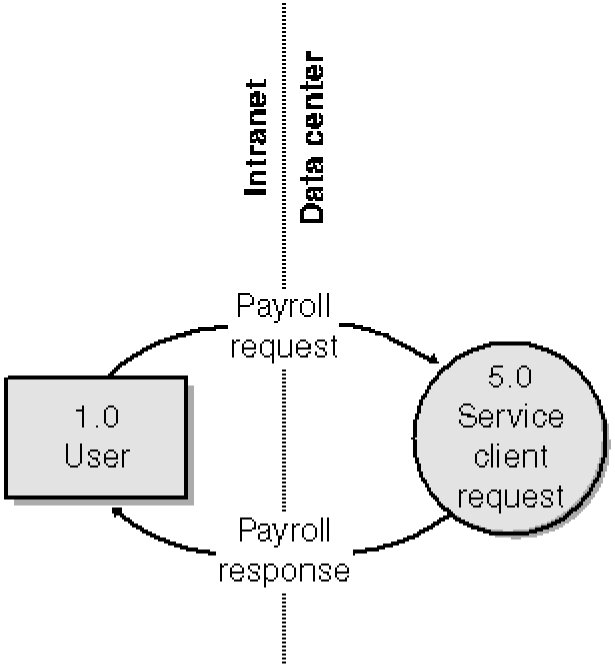

Attack # 1 View confidential payroll data on the wire (I)
and
1.1 HTTP traffic is unprotected
1.2 Attacker views traffic
1.2.1 Sniff network traffic with protocol analyzer
1.2.2 Listen to router traffic
1.2.3 Compromise switch
and
1.2.2.1 Router is unprotected
1.2.2.2 Compromise router
1.2.2.3 Guess router password
1.2.3.1 Various switch attacks

Vulnerability = Information disclosure

Step 3 (cont)

- In practice text descriptions are necessary because trees tend to be very large

1.0 View confidential payroll data on the wire
  - 1.1 HTTP traffic is unprotected (AND)
  - 1.2 Attacker views traffic
    - 1.2.1 Sniff network traffic with protocol analyzer
    - 1.2.2 Listen to router traffic
      - 1.2.2.1 Router is unpatched (AND)
      - 1.2.2.2 Compromise router
      - 1.2.2.3 Guess router password
    - 1.2.3 Compromise switch
      - 1.2.3.1 Various switch attacks



- Typically there will be too many trees to generate, so to reduce the overall number/work
  - ☞ confirm that the attack really makes sense for a given component/interaction
  - ☞ try to re-use attack trees

Step 4: Rank vulnerabilities

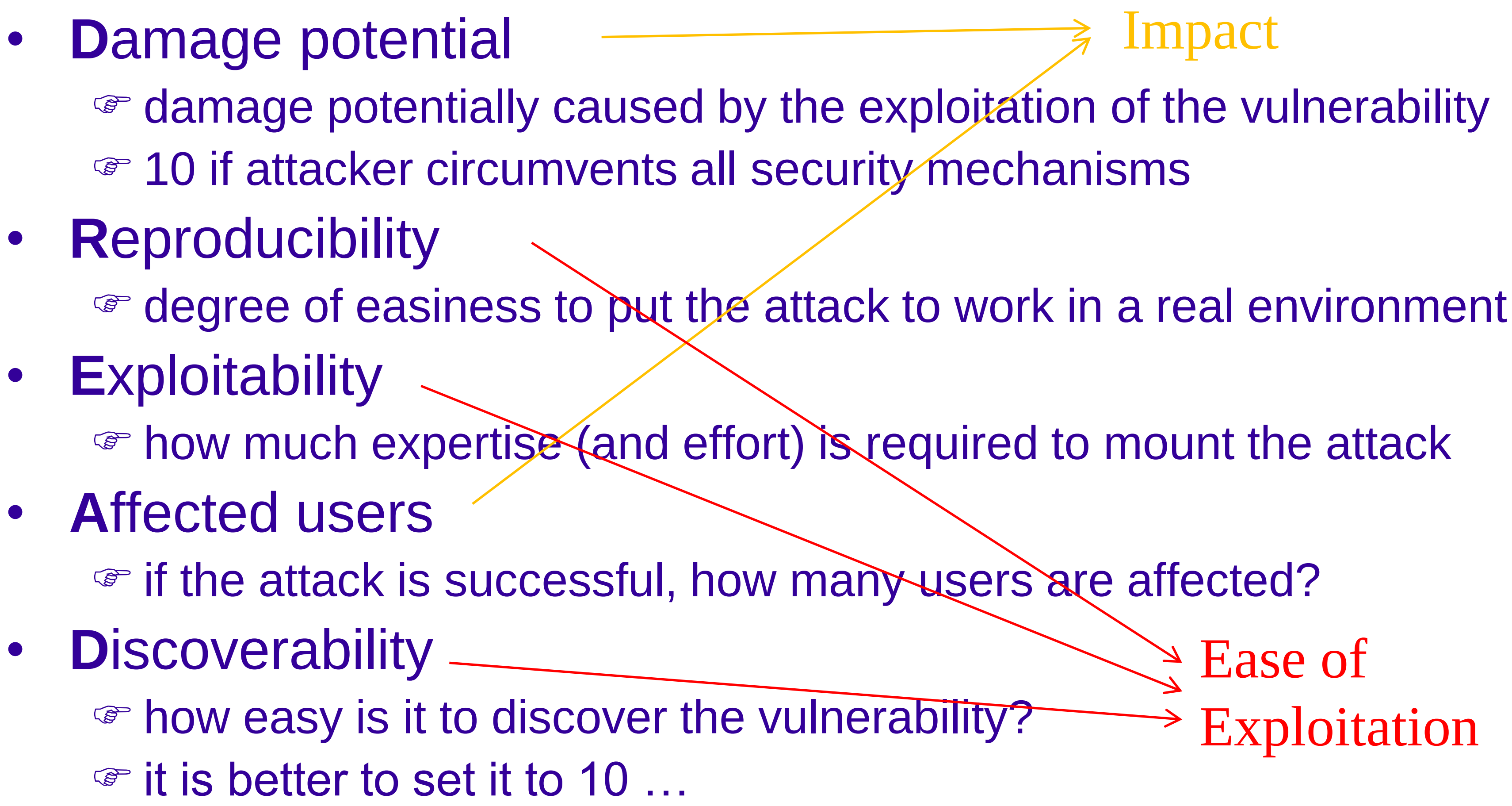
- The goal is to help prioritize the vulnerabilities that must be addressed first (in some cases, one can choose not to correct some vulnerabilities)

- Risk is a very relevant cost metric that can serve as basis for the creation of a ranking
  - ☞ *risk = probability of successful attack x impact*
  - ☞ *probability of successful attack = level of threat x degree of vulnerability*

- Example: Microsoft's DREAD methodology
  - ☞ gives an estimate for the risk
  - ☞ average of 5 parameters in range 1-10

DREAD

Risk = average of the DREAD values

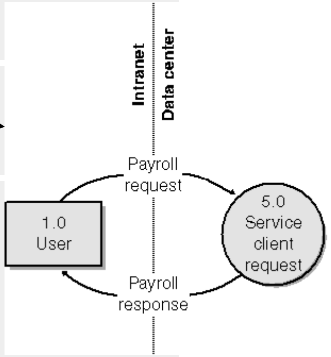


- **D**amage potential
  - ☞ damage potentially caused by the exploitation of the vulnerability
  - ☞ 10 if attacker circumvents all security mechanisms
- **R**eproducibility
  - ☞ degree of easiness to put the attack to work in a real environment
- **E**xploitability
  - ☞ how much expertise (and effort) is required to mount the attack
- **A**ffected users
  - ☞ if the attack is successful, how many users are affected?
- **D**iscoverability
  - ☞ how easy is it to discover the vulnerability?
  - ☞ it is better to set it to 10 …

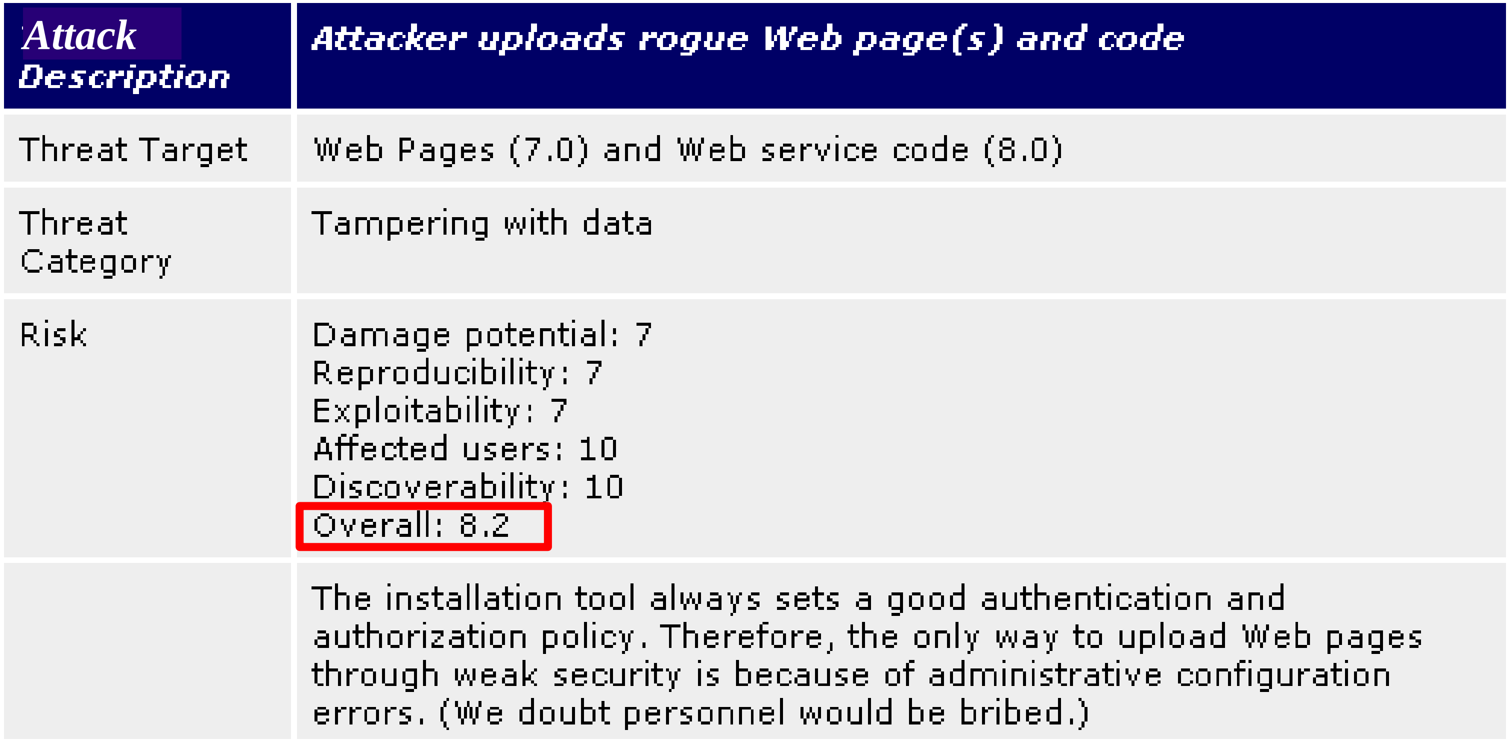
Example attack 1

Attack Description	*Malicious user views confidential on-the-wire payroll data*
Threat Target	Payroll Response (5.0 → 1.0)
Threat Category	Information disclosure
Risk	Damage potential: 8 Reproducibility: 10 Exploitability: 7 Affected users: 10 Discoverability: 10 Overall: 9
Comments	Most likely attack is from rogue user using a protocol analyzer, because it's an easy attack to perform; the attack is passive and cheap in terms of time, effort, and money. The switch threat is important because many people think switched networks are secure from sniffing attacks when in fact they are not. If you think they are, take a look at "Why your switched network isn't secure" at http://www.sans.org.

stride



Example attack 2



Attack Description	Attacker uploads rogue Web page(s) and code
Threat Target	Web Pages (7.0) and Web service code (8.0)
Threat Category	Tampering with data
Risk	Damage potential: 7 Reproducibility: 7 Exploitability: 7 Affected users: 10 Discoverability: 10 Overall: 8.2
	The installation tool always sets a good authentication and authorization policy. Therefore, the only way to upload Web pages through weak security is because of administrative configuration errors. (We doubt personnel would be bribed.)

Benefits of attack modeling

- helps reduce risk by supporting the removal of vulnerabilities (starting with the most relevant)
- helps find bugs (besides vulnerabilities)
- helps understanding the application, both in terms of the attacks that can occur and also the available security measures
- documents the application for other teams that may use it as a component
- helps testers define what has to be tested

Problems and alternatives

- Attack modeling does not scale
  - ☞ "The practicality of attack trees to characterize attacks on real-world systems depends on being able to reuse previously developed patterns of attack." Moore et al.
- Example pattern: Buffer overflow attack (i.e., subtree)
  - AND
  - 1. Identify vulnerability in the target
  - 2. Write code to be executed in the target
  - 3. Prepare input to cause the overflow
  - 4. Inject input to cause the overflow
- Try to do some of the work automatically with the help of some tool
  - ☞ Example: *Topological analysis of vulnerabilities*, does attack modeling automatically for a distributed system

Other risk assessment methodologies

- OCTAVE
  - ☞ Information security risk management
  - ☞ Developed at the CERT Coordination Center
- OWASP CLASP
  - ☞ Comprehensive Lightweight Application Security Process
  - ☞ structured approach for moving security concerns into the software development cycle

CODE REVIEW

Introduction to code review

- *Code review* or *code auditing* or *source code auditing* or…
- Phases
  1. **Pre-assessment**: planning, defining scope of review, information collection -- some similarities to info gathering
  2. **Code review**: main phase
  3. **Documentation and analysis**: creation of documentation, risk analysis -- next slide
  4. **Remediation support**: assist those that have to use the results of the review
- Using a vulnerability taxonomy (like STRIDE) and a vulnerability rank (like DREAD) is again important

Documentation

- For each vulnerability found it is necessary to provide the following information
  - ☞ Attack (e.g., brute-force login)
  - ☞ Affected component (e.g., login component)
  - ☞ Module details (e.g., login.php, lines 76-89)
  - ☞ Vulnerability class (e.g., authentication bypass)
  - ☞ Description
  - ☞ Result
  - ☞ Prerequisites
  - ☞ Business impact
  - ☞ Proposed remediation
  - ☞ Risk (e.g., in terms of DREAD)

Discussion

- In project 1 you did a manual code review

- How did you do it?
  - ☞ where should you start?
  - ☞ how should you proceed with the review?

- If you have hundreds of thousand of lines of code, you need a good review process
  - ☞ Strategies
  - ☞ Tactics

Code review strategies

Understanding the Code

- systematically analyze the source code to *understand in detail the application* and find vulnerabilities
  - - requires reading the code
  - + provides good knowledge of the code
  - + allows to find subtle vulnerabilities

Vulnerability Classes

- create list of *potential security issues* (e.g., buffer overflows) and then look for them in the code
  - + fastest way of finding known types of vulnerabilities
  - - does not encourage strong knowledge of the application

Design Problems

- analyze medium/high-level logic to *identify design flaws*

1. Understanding the code

Follow the Malicious Inputs

- Start at a data entry point and trace its flow on code, looking for security issues
  - *difficult to go off track*
  - *time consuming, easy to overlook issues*
  - *difficult with object-oriented code*

Analyze a module (or algorithm)

- Read code line by line taking notes of possible issues; used by many experienced reviewers
  - *not jumping around avoids distractions, and since the code in a file tends to be cohesive (e.g., network interface) it may be easier to review*
  - *hard, mentally taxing, lack of context of how the module is accessed*

Trace injection hits

- Use data of black-box testing (fuzzing, attack injection) hit information (e.g., crash) as a guide to optimize first tactics (follow the malicious inputs)

2. Vulnerability Classes

General candidate points approach

- Pick lowest level routines that handle relevant resources, look for possible vulnerabilities, when found backtrack to see if reachable from input
  - *good coverage of known vulnerabilities*
  - *hard to go off track*
  - *lower comprehension of the code, and is limited to known issues*

Simple candidate points

- Use simple tools (e.g., grep) to find candidate vulnerabilities by searching for well-known code patterns
  - *simple*
  - *limited*

Warnings candidate points

- Similar to the third strategy of the previous slide, as it uses data of black-box testing (fuzzing), but instead of starting from the problematic entry point (input), it starts from the lines that might have caused the bug to be exploited

Code review tactics

- Review carefully error-checking branches and small branches
- Analyze dependencies between modules /functions/objects (instead of only inside them)
- Reread the code (don’t read only once)
- Desk-checking (do a table with variables’ values and see how they evolve through the code)
- Test cases (define test values and use them in pieces of code)

Code review tools

- Source code navigators
  - ☞ similar to IDEs but focused on reading/following code, not writing
  - ☞ some functions: cross-referencing, text search, syntax highlighting, graphing (e.g., call trees)
  - ☞ Cscope, Ctags, Source Navigator, Code Surfer, Understand
- Binary navigators
  - ☞ IDA Pro, BinNavi
- Debuggers
- Attack injectors / fuzzers

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 12)

Other references:

- M. Dowd, J. McDonald, J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*, Addison-Wesley Professional, 2006 (see chapter 2 and 4)
- A. Shostack, *Thread Modeling: Designing for Security*, Wiley, 2014