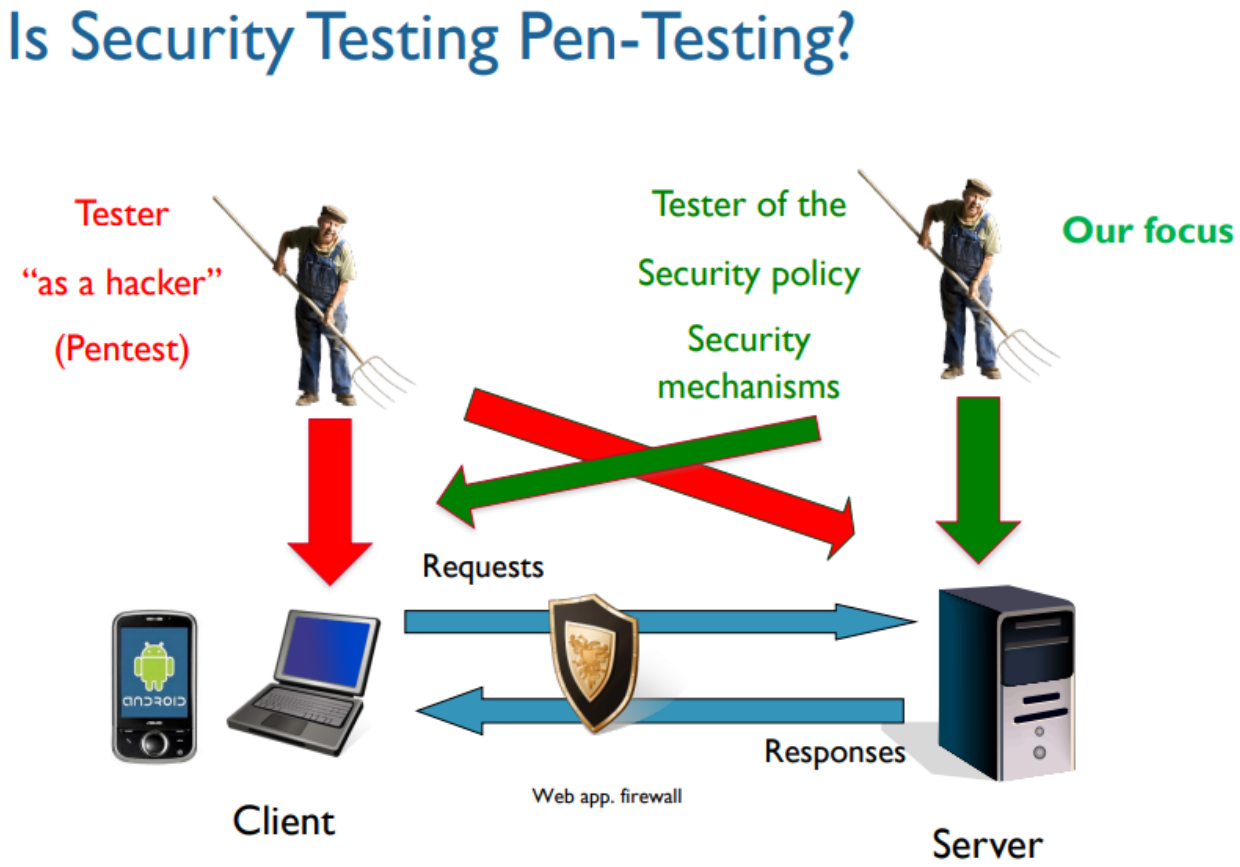
C11 Security Testing

1. General Principles

1.1 Security vs Functional vs Penetration Testing

- Functional testing is about proving some features work as specified
- Security testing is about checking that some features appear to fail, demonstrating that
  - Tester cannot **spoof** another user's identity
  - Tester cannot **tamper** with data
  - There is no **repudiation** issues
  - Tester cannot view **information** he does not have access to
  - Tester cannot **deny** services of the system
  - Tester cannot **elevate privileges**



1.2 Building Security Test Plans from Threat Model

- Rigorous testing plan derived after a threat model (C2) is developed
  1. Decompose application into fundamental components
  2. Identify component interfaces

3. Rank interfaces by potential vulnerability
4. Ascertain the data structures used by each interface
5. Perform testing to find security problems by injecting mutated data

Decomposing of Application

- Threat models can aid the testing process by providing three valuable items:
  - List of components in the system (which needs testing)
  - Threat types to each component (STRIDE)
  - Threat risks (DREAD, CVSS)

Identifying component interfaces

- Determines interfaces exposed by each component, best found in functional specification
- Examples include
  - Broadcast receivers
  - HTML forms, HTTP requests, URLs, SOAP requests
  - TCP and UDP sockets, wireless data
  - Shared memory, LPC and RPC interfaces
  - Database access technologies
  - Dialog boxes, console inputs, command line arguments, files, microphone, hardware devices

Ranking by potential vulnerability

- Prioritises which interface needs testing first, may come from threat model's risk ranking; but add more granularity and accuracy for testing purposes
- Use a simple point-based system to determine relative vulnerability

Interface Characteristic	Points
The process hosting the interface or function runs as a high-privileged account such as root	2
The interface handling the data is written in a higher-level language than C/C++, such as C#, Java, Python, etc.	-2
The interface handling the data is written in C/C++	1
The interface takes arbitrary-sized buffers or strings	1
The recipient buffer is stack-based	2
The interface has no or weak access control mechanisms	1
The interface or resource has good access control mechanisms	-2
The interface does not require authentication	1
The feature is installed by default	1
The feature has already had security vulnerabilities	1

Ascertain data structures used by each interface

- Analyse the specifications or any available information of a given interface and identify its data structures (input parameters and valid input domains) which are to be modified later to expose security bugs

Testing to find security problems based on STRIDE

- Formulate test plans based on threat model, and every threat model must have a test plan outlining one or more tests; which should be specified with the expected successful result (**test oracle**)
- Existence of vulnerabilities in components that are used but not directly controlled should also be determined

Threat Type	Testing Techniques
Spoofing identity	• Attempt to force the application to use no authentication; is there an option that allows this, which a non-admin can set? • Try forcing an authentication protocol to use a less secure legacy version? • Can you view a valid user's credentials on the wire or in persistent storage? • Can "security tokens" (for example, a cookie) be replayed to bypass an authentication stage? • Try brute-forcing a user's credentials; are there subtle error message changes that help you attempt such an attack?
Tampering with data	• Attempt to bypass the authorization or access control mechanisms. • Is it possible to tamper with and then rehash the data? • Create invalid hashes, MACs, and digital signatures to verify they are checked correctly. • Determine whether you can force the application to roll-back to an insecure protocol if the application uses a tamper-resistant protocol such as SSL/TLS or IPSec.
Repudiation	• Do conditions exist that prevent logging or auditing? • Is it possible to create requests that create incorrect data in an event log? For example, including an end-of-file, newline, or carriage return character in a valid request. • Can sensitive actions be performed that bypass security checks? (using "Spoofing identity" and "Tampering with data")
Information disclosure	• Attempt to access data that can be accessed only by more privileged users. This includes persistent data (file-base data, registry data, etc.) and on-the-wire data. Network sniffers are a useful tool for finding such data. • Kill the process and perform disk scavenging, looking for sensitive data that is written to disk. You may need to ask developers to mark their sensitive data with a common pattern in a debug release to find the data easily. • Make the application fail in a way that discloses useful information to an attacker. For example, error messages.

Denial of service	Probably the easiest threats to test! • Flood a process with so much data it stops responding to valid requests. • Does malformed data crash the process? This is especially bad on servers (e.g. loss of data). • Can external influences (such as reduced disk space, memory pressure, and resource limitations) force the application to fail?
Elevation of privilege	• Spend most time on applications that run under elevated accounts, such as SYSTEM services • Can you execute data as code? • Can an elevated process be forced to load a command shell, which in turn will execute with elevated privileges?

(not examinable)

Generic test plan

- Given: an interface to test, as well as input parameters and valid input domains
- Choose test inputs (malformed inputs) for the input parameters that suit the threat model
- Determines expected outputs for chosen inputs (test oracle)
- Constructs tests with chosen inputs

1.3 Test Oracle

- A mechanism that determines whether software is executed correctly for a test case and contains two essential parts
  - Information that represents expected output
  - Procedure (manual / automated) that compares the oracle information with the actual output
- Steps
  1. Input X into code
  2. Input X into oracle
  3. Compare output of code with output of oracle, if same output, then test was successful, unsuccessful otherwise
- Oracle types
  - Expert human being
  - Assertions (`assertEquals(x, 34)`)
  - Specification and documentation
  - Other programs (that uses a different algorithm to value the same expression as product under test)
  - Heuristic code that provides approximate results or exact results for a set of test inputs
  - Consistency oracle that compares the results of one test execution to another for similarity (regression testing)

Testing approaches

- Black Box testing
  - Functional test case selection criteria (focuses on external behaviour)
  - Representations for consider combinations of events / stats (mapping of usage scenarios, cause-effect diagrams, decision table)
- White Box testing
  - Coverage based
  - Fault-based (mutation testing)
  - Failure-based (use representations created by symbolic execution)

Classifying output responses

- (V) Valid responses (expected outputs) - malformed inputs are adequately processed by server
  - V1 - server acknowledges invalid request and provides and explicit message regarding violation
  - V2 - server produces generic error messages
  - V3 - server ignores request completely
- (F) Faults & Failures - malformed input cause abnormal software behaviours, when the code behind the interface being tested fails to handle the input

2. Black / Grey Box (fuzzing)

- Building security test cases to exercise the interfaces by using data mutation (randomization to construct malformed inputs)
- Perturb the environment such that the code handling the data that enters an interface behaves in an insecure manner

2.1 Mutation Operators

Container

- Resource already exists / not exist
  - How does application react if a file does not exist if it requires the file to exist
- Deny access or restricted access to the container
  - Set Deny / Restrict access control entry on the object prior to running the test
- Name or Link
  - How does application react if container exists but has a different name or the other way around
- Tested threat types: **Information Disclosure, Tampering with Data**

Data

- Two characteristics (content and size), which should be maliciously manipulated

- **Size**
  - Zero length
    - Effect of empty input field
  - Long (input too long)
    - Domain of buffer overrun attacks
  - Short (input too short)
    - When applications expect a fixed size but gets less data
- **Content**
  - Null (Missing data)
  - Zero
  - Wrong sign
  - Wrong type (ANSI vs unicode)
  - Random input
  - Valid + Invalid data (content consists of valid data which helps to get past checks but also contains invalid data at the same time, e.g. dd/mm/yyyyABCDE)
  - Out of bound

Network

- Replay - replay messages from old authentication request (cookies)
- Out of sync - how does a component deal with inputs that arrive out of order
- High volume - how does system react to DDoS
- Partially correct protocol runs - messages follow protocol partially correct

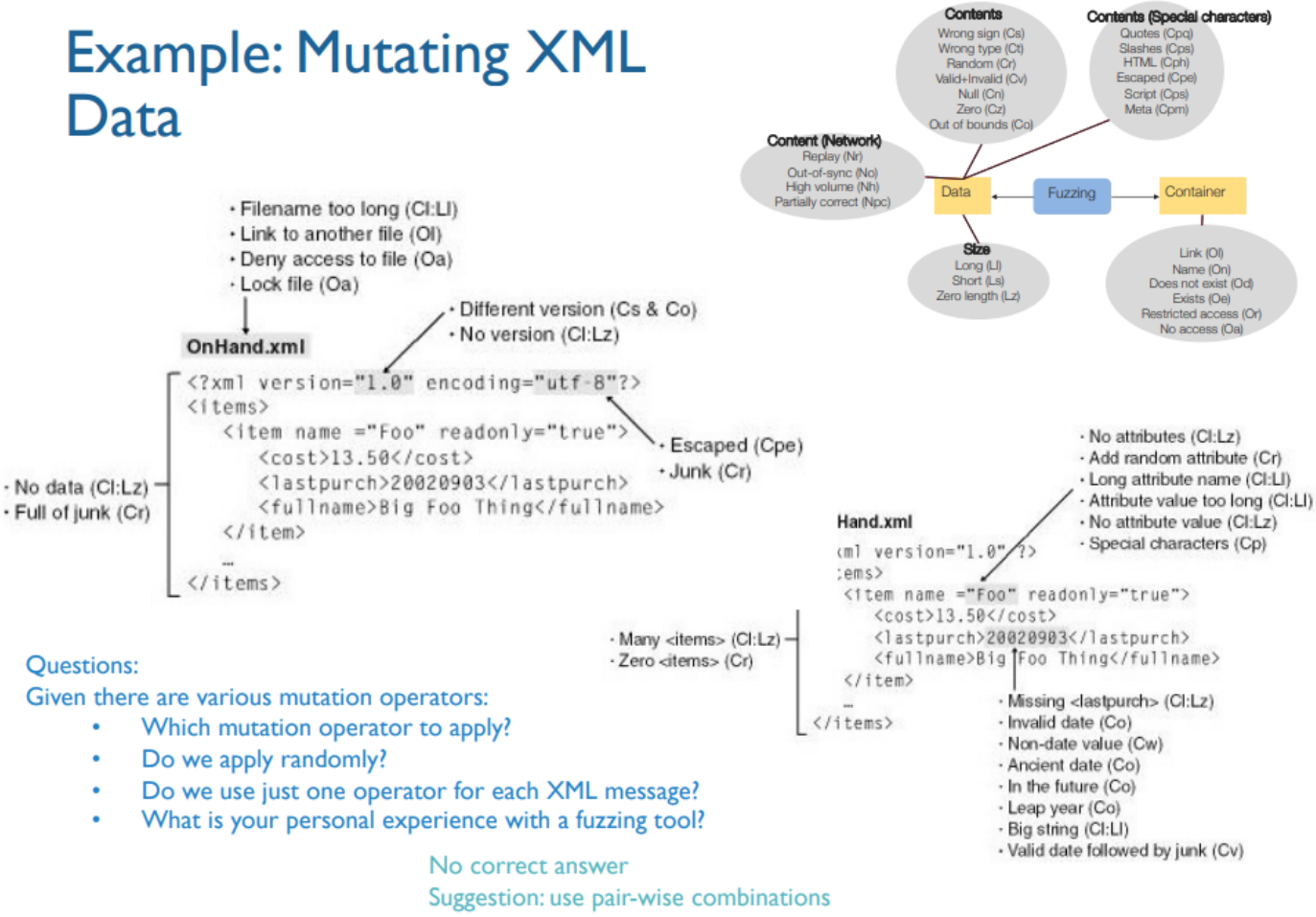
Special Characters

- Use characters that have special semantics to an interpreter
  - Quotes
  - Slashes
  - Escaped data
  - HTML
  - Script
  - Character encoding

- Meta-characters etc

Character	Comments
‘ ’’	String delimiters
\	Backslash (escape character)
( ) [ ] { }	brackets
<!– and -->	HTML and XML comment operators
--	SQL comment operator
\n and \r or 0x0a and 0x0d	Newline and carriage return
\t	Tab
0x04	End of file
0x7f	Delete
0x00	Null bytes
< and >	Tag delimiters
* ? _	Wildcards
, ; & +	others

Example: Mutating XML Data



Questions:
Given there are various mutation operators:

- Which mutation operator to apply?
- Do we apply randomly?
- Do we use just one operator for each XML message?
- What is your personal experience with a fuzzing tool?

No correct answer
Suggestion: use pair-wise combinations

Example: Mutating Web Form Inputs

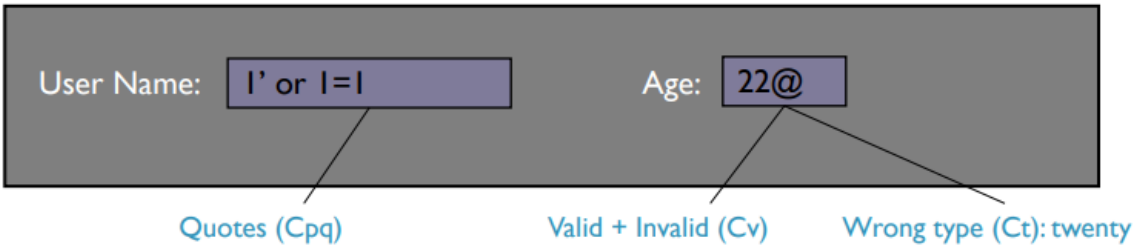
- Valid Web Form Inputs:
  - User name should be plain text only
  - Age should be a digit

Test Plan:

- Web form interface to test
- Identify input parameters and their valid domains

- Choose test inputs (malformed inputs)
- Test Oracle?

- Mutated Web Form Inputs:



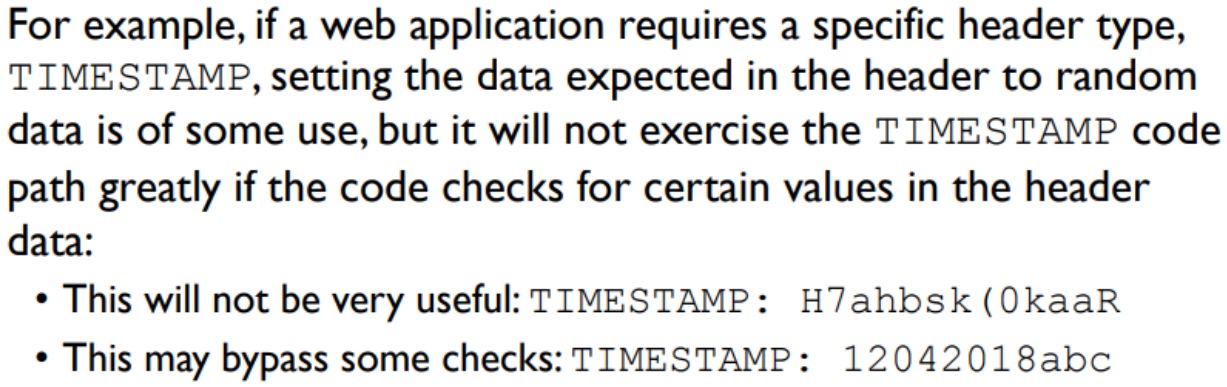
Randomization

- Random inputs test how component handles unexpected inputs which are usually caught by simple input checks

- Valid + invalid operator

2.2 Domain Knowledge

- Use when tester wants to mutate data but still want application to exercise some path properly
  - By mutating from valid structure of data that tester knows rather than randomly



2.3 Practical Problems with Automated Fuzzing

- Can only detect simple faults or vulnerabilities
- Requires significant running time since it is not efficient
- Usually achieves poor code coverage
- Does usually not find logic flaws
  - Malformed data leads to crashes and not logic flaws
- Harder to apply to types of vulnerability that do not cause program crashes

3. White-box (symbolic execution)

- Analysis of programs by tracking symbolic rather than actual values and is used to reason about all the inputs that take the same path through a program
- Builds predicates that characterise
  - Conditions for executing paths
  - Effects of execution on program state
- Random testing - generate random inputs and execute program on concrete ones

Predicate

- Boolean value statement that is either true / false depending on values of variables

Execution Path

- Can be seen a binary tree (because you either take the branch or you don't) with possible infinite depth
- Each edge represents the execution of a sequence of non-conditional statements between a conditional statement

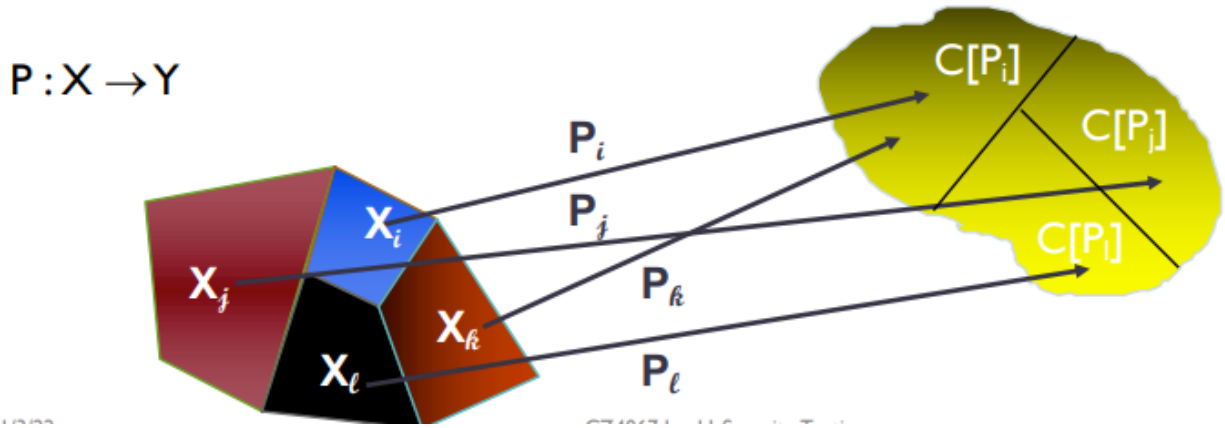
- Each path will represent an equivalence class of inputs (inputs that lead to the same outcome)

2.1 Properties

- Uses symbolic values for input variable
- Builds predicates that characterises the conditions under which execution paths can be taken
- Collects symbolic path constraints
- Uses constraint solvers to check if an answer exists and the branch can be taken
- Negates path constraints to take the other side of the condition

2.2 Representing Computation Symbolically

- **Symbolic names (X_i)** - input values
- **Path value (PV)** - describes value of variable in terms of symbolic name
- **Computation ($C[P_i]$)** - described by path values of outputs for the path
- **Path condition (PC)** - describes the domain of the path and is the conjunction of the interpreted branch conditions
- **Domain of path (D[P])** - set of input values that satisfy the PC for the path



2.3 Constraint Solving

- Mathematically determine if a set of constraints can be solved
- Linear equations `if (3 + x < 6) do a; else do b`
  - Constraint solver would require x $<$ 3 to cover if branch and x $\geq$ 3 to cover else branch
- Loop unrolling can be done to handle loops

2.4 Problems

- Expensive to create program representations
- Expensive to reason about expressions
- Problems with function calls where calling of contexts need to be kept track of
- Loops often are unrolled to a certain depth rather than having termination and invariants dealt with

- Aliasing which leads to massive blow-up in the number of paths

4. Test selection / prioritisation

- Problems
  - Huge number of test cases
  - High testing cost
  - Budget and time constraints pose barriers to testing process
- Proposals
  - Test in a best effort basis
  - Maximise test effectiveness per utilised test
  - Stop testing when constraints are met
  - Consider removing the features that were not tested

4.1 Combinatorial Testing

- System under test (**SUT**) may have many input parameters each having many possible values. So interaction between parameters need to be tested to test a system properly (impossible due to large number of combinations)
  - Input spaces that are more likely to lead to security bugs should be found first (Pareto's Law states that 80% of bugs come from 20% of modules)

4.2 Pairwise Testing

- Combining input parameters in a systematic way and test at least one combination for every combination of each two-parameters
- Most common bugs are generally triggered by either a single input parameter or an interaction between pairs of parameters.