NANYANG TECHNOLOGICAL UNIVERSITY

Lecture 11: Security Testing

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

Finding the Right Balance

"You can't test it, you can't ship it."

"Exhaustive testing is impossible."

"Testing shows presence of defects."

"It's the testing process that determines whether the product is secure."

Some contradicting quotes

Agenda

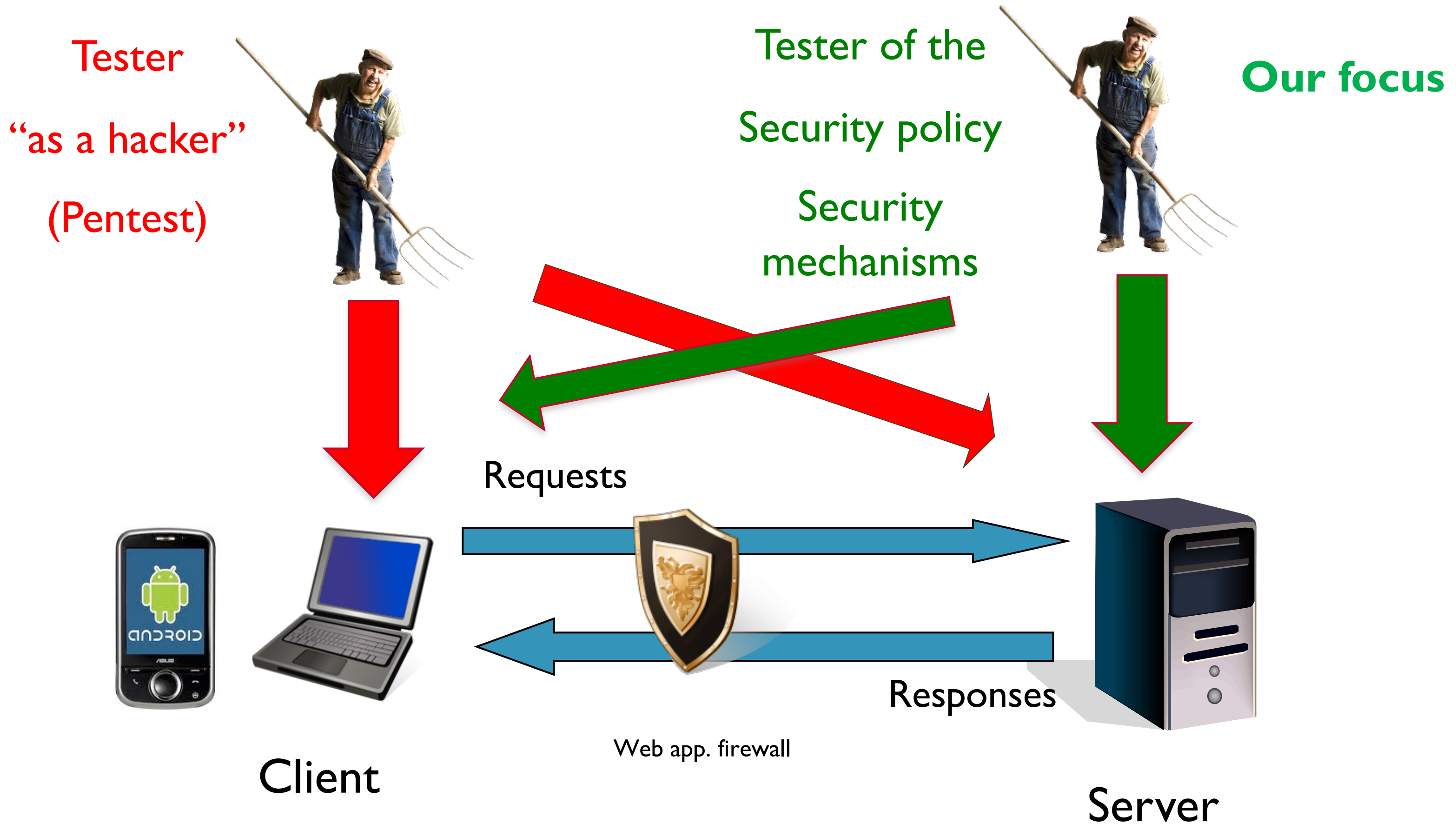
- Security testing: general principles
- Black/grey-Box (fuzzing)
- White-box (symbolic execution)
- Test selection/prioritization
- Security testing in action

Security Testing vs Functional Testing

- Functional testing is about proving some features work as specified
- Security testing is about checking that some features appear to fail, i.e., it involves demonstrating that
  - the tester cannot Spoof another user's identity,
  - the tester cannot Tamper with data,
  - there is no Repudiation issues,
  - the tester cannot view Information he shouldn't have access to,
  - the tester cannot Deny services of the system, and
  - the tester cannot Elevate privileges

STRIDE

Is Security Testing Pen-Testing?



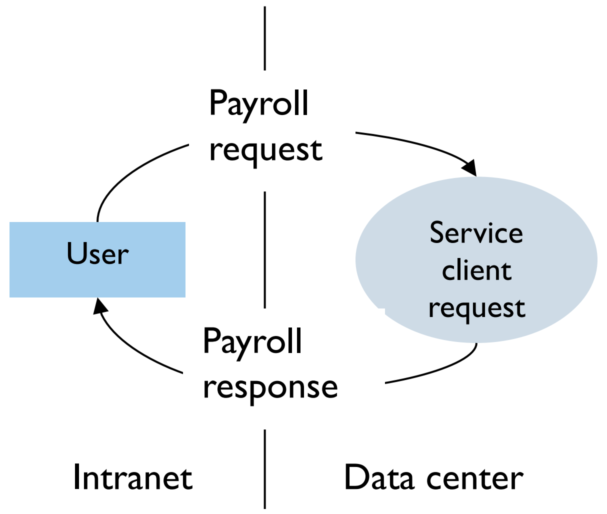
Building Security Test Plans from a Threat Model

- Rigorous testing plan derived based on the threat model (Lecture 2):
  1. Decompose the application into its fundamental components.
  2. Identify the component interfaces.
  3. Rank the interfaces by potential vulnerability.
  4. Ascertain the data structures used by each interface.
  5. Perform testing to find security problems by injecting mutated data.

*A testing process used at Microsoft

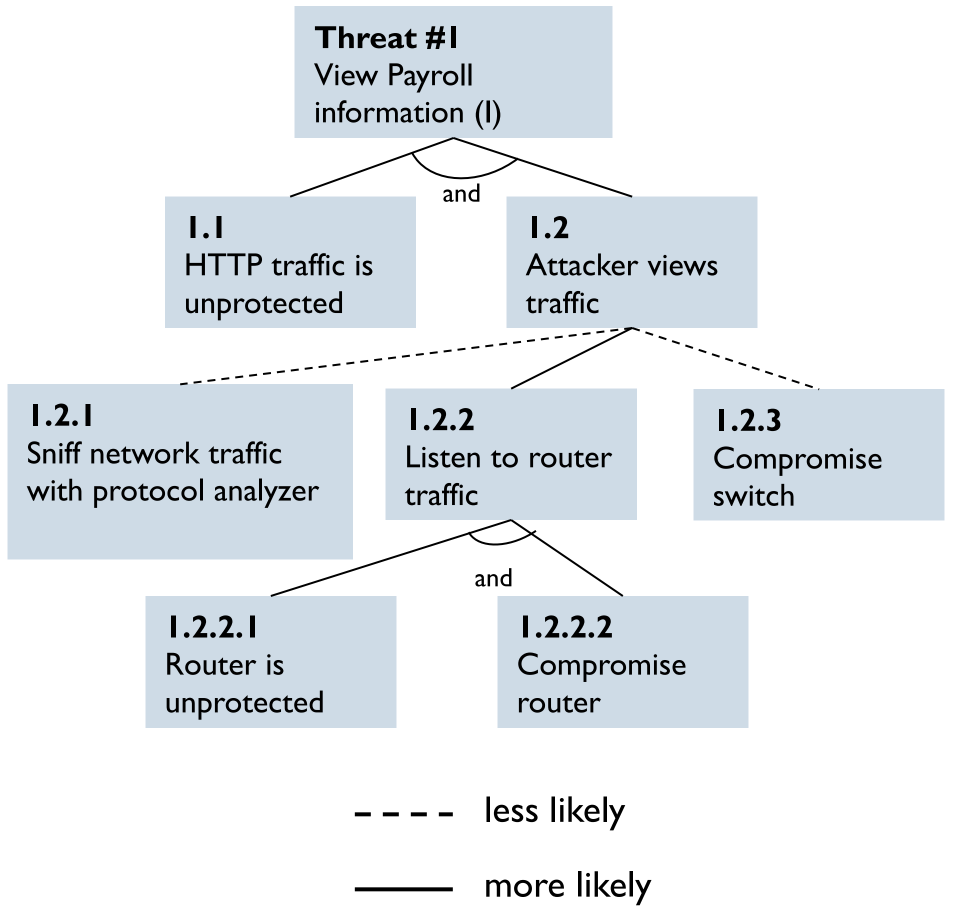
Recap: Threat Model

- The payroll data that flows between the user (an employee) and the computer systems inside the data centre
- It's sensitive data, confidential between the company and its employees— you don't want a malicious user looking at someone else's payroll information.
- This is an example of an information disclosure threat from STRIDE



Example: Threat Tree

- Show a number of ways an attacker can view the data.
- The top box is the ultimate threat; the boxes below are the steps involved to make the threat an attack.
- Record the type of threat based on STRIDE or any applicable model


Threat #1 View Payroll information (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
less likely
more likely

1. Decompose the Application

- Threat models are not just a design tool
- It can aid the testing process by providing three valuable items:
  - The list of components in the system – components that need testing (features or resources that need to be tested)
    - E.g. in a mobile app, services are components that implement certain app features
  - The threat types to each component (STRIDE or similar)
  - The threat risk (DREAD, CVSS, or similar)

2. Identify the Component Interfaces

- Determine the interfaces exposed by each component
- Best place to find exposed interfaces is in functional specifications
- Example interfaces 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 such as USB, Bluetooth

3. Rank the Interfaces by Potential Vulnerability

- To prioritize which interfaces need testing first
- May come from threat model's risk ranking; but add more granularity and accuracy for testing purposes
- The process of determining the relative vulnerability of an interface is to use a simple point-based system

- Note: if the list of interfaces is large and you have not sufficient time to test some of them adequately, consider removing those interfaces and the features behind them from the product

Example: Points to Attribute to Interface Characteristics

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

4. Ascertain the Data Structures Used by Each Interface

- Given an interface, analyze its specifications or any available information
- Identify its data structures – input parameters and their valid input domains
- Later you will modify them to expose security bugs

Example

Interface	Data
Sockets, RPC, Named Pipes	Data arriving over the network
Files	File contents
Registry	Registry key data
Active Directory	Nodes in the directory
Environment	Environment Variables
HTTP data	HTTP headers, form fields, query strings, XML payloads, SOAP data and headers
Command line arguments	Data in argv[] for C/C++, string[] args array in C# applications

5. Testing to Find Security Problems based on STRIDE

- Formulate test plans based on the threat model. (The tables in the next three slides list some general guidelines)

- Every threat in the threat model must have a **test plan** outlining one or more tests; a test should be specified with the expected successful result (test oracle)

- You should also determine whether vulnerabilities (CVE entries) exist in components that you use but do not directly control, such as DLLs, class libraries, call backs

Testing STRIDE I

Not examinable!
***you are still expected to know the threat types**

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.

Testing STRIDE 2

Not examinable!
*you are still expected to know the threat types

Threat Type	Testing Techniques
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.

Testing STRIDE 3

Not examinable!
***you are still expected to know the threat types**

Quiz: What is the threat type of XPath injection?

Threat Type	Testing Techniques
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?

From CZ2006 Software Engineering course

A Generic Test Plan

- Given: an interface to test, its input parameters, their valid input domains (and equivalence classes)
- Chooses test inputs (malformed inputs) for the input parameters that suit the threat model
- Determines expected outputs for chosen inputs (test oracle)
- Constructs tests with the chosen inputs
  - e.g., write a Perl script that generates XML messages and sends them to the Web service under test

Test Oracle

- A test oracle is a mechanism that determines whether software executed correctly (or incorrectly) for a test case
- A test oracle contains two essential parts:
  - Oracle information that represents expected output
  - An oracle procedure (manual or automated) that compares the oracle information with the actual output
- How it works:
  - Step 1: input X into the code
  - Step 2: input X into the oracle
  - Step 3: compare the output of the code with the output of the oracle
  - If the outputs are the same, the test was successful. Otherwise, there is a fault in the code

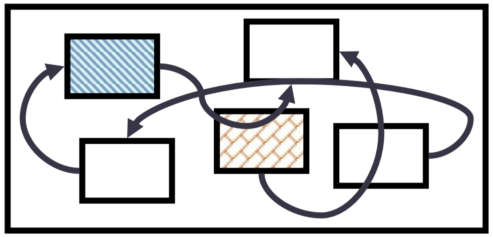
Test Oracles

Common types of oracles:

- An expert human being
- Assertions: `assertEquals(x, 34)`
- Specifications and documentation (e.g., invariants)
- Other programs (e.g., program that uses a different algorithm to evaluate the same expression as the product under test)
- A heuristic oracle that provides approximate results or exact results for a set of test inputs
- A consistency oracle that compares the results of one test execution to another for similarity (e.g., regression testing)

Approaches to Testing

- Black Box Testing Strategies
  - Functional test case selection criteria (focus on external behavior)
  - Representations for consider combinations of events/states (map out usage scenarios, cause-effect diagram, decision table)
- White Box Testing Strategies
  - coverage-based
  - fault-based (e.g., mutation testing)
  - failure-based (domain and computation-based, use representations created by symbolic execution)

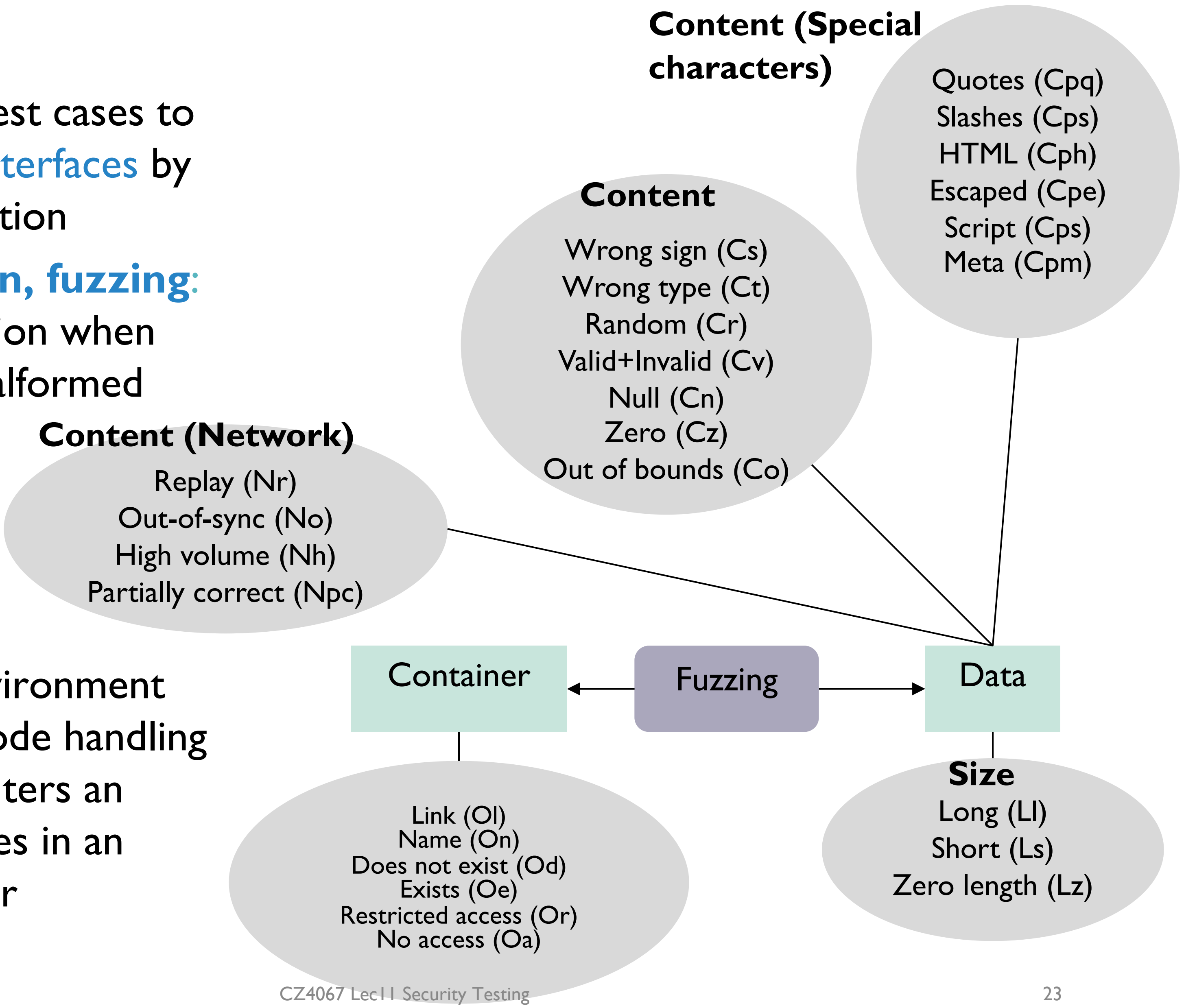


Fuzzing

Mutating inputs to observe its effects on security properties

Fuzzing

- Build security test cases to exercises the interfaces by using data mutation
- **Data mutation, fuzzing**: use randomization when constructing malformed inputs
- Perturb the environment such that the code handling the data that enters an interface behaves in an insecure manner



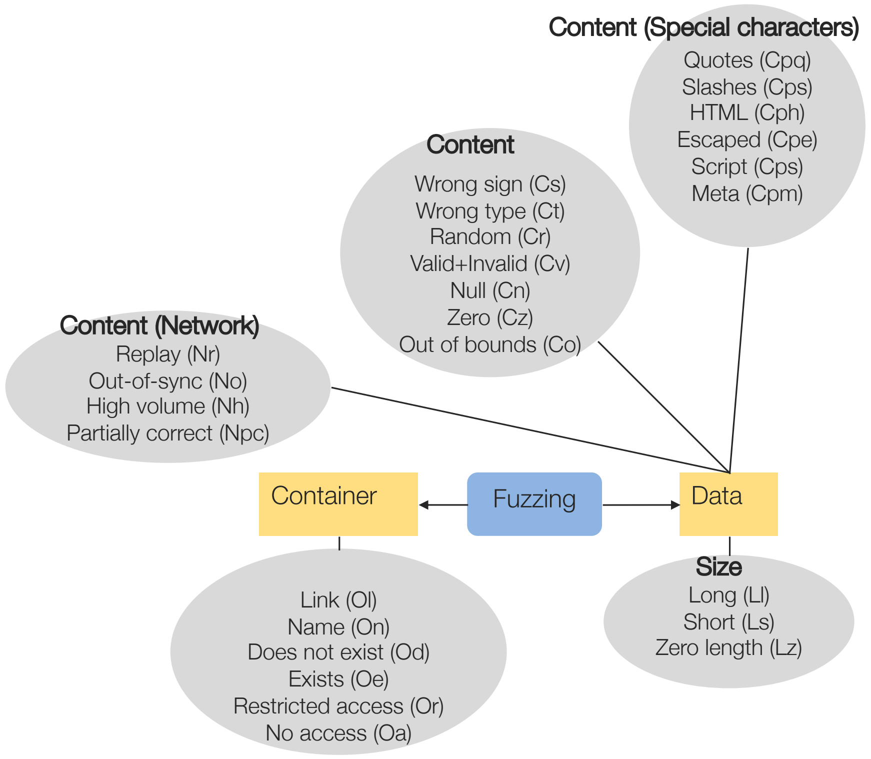
Mutation Operators: Container

- The resource already exists or not exist: Imagine that your application requires that a file already exists— how does the application react if it does not exist? Does it take on an insecure default?
- 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 the application react if the container exists but the name is different? Or if the name is valid but the name is actually a link to another file?

Threat types that can be tested: information disclosure, tampering with data

Mutation Operators: Data

- Data has two characteristics – the nature of the content and the size of the data – both of which should be maliciously manipulated



Mutation Operators: Size

Threat types?

- Zero length
  - What happens if an input field is empty?
  - Issue e.g. with XML documents
- Long: Too long
  - Typical domain of buffer overrun attacks
- Short: Too short
  - What happens if the application expects a fixed size input but gets less data?
  - E.g. Sun tarball vulnerability (memory residues)

Mutation Operators: Content

- Null: missing data
- Zero
- Wrong sign
  - Brought down Ariane 5 http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html
- Wrong type
  - E.g. ANSI or Unicode character
- Random input: arbitrary data
- Valid + invalid data
  - Valid data to get past input checks so that invalid data may actually be processed; e.g. 09/12/2018jk16
- Out of bound: e.g. future dates
  - Crashing Tetris: https://www.csoonline.com/article/2136265/data-protection/how-to-crash-an-in-flight-entertainment-system.html

Mutation Operators: Network

Threat types?

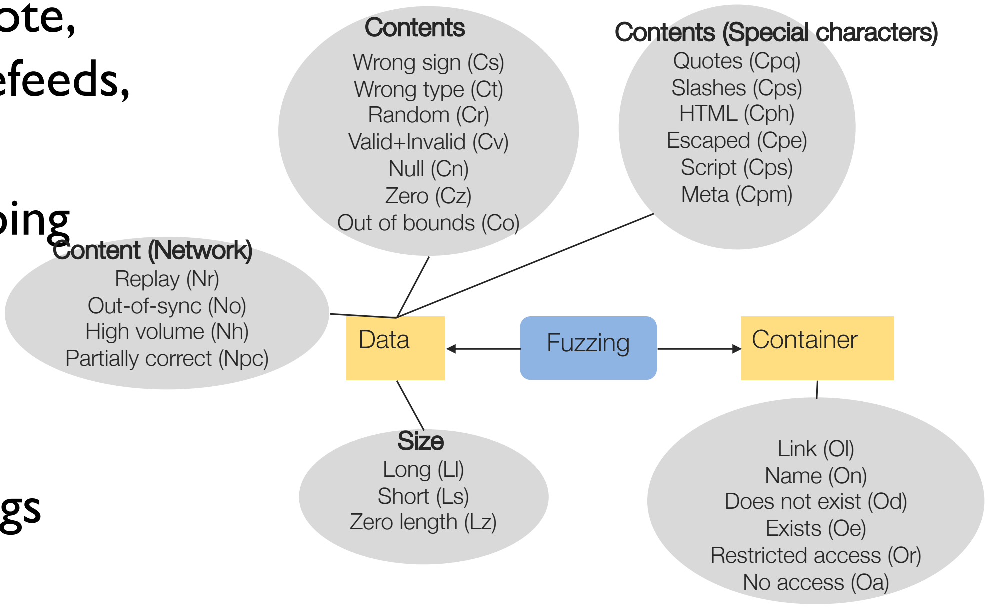
- Replay: e.g. replay messages from an old authentication request (cookie)
- Out of sync
  - How does the component deal with inputs that arrive out of order?
  - An application may perform security checks only on `Data1`, which allows `Data2` & `Data3` to enter the application unchecked. Some firewalls/IDS have been known to do this.
- High volume: How does the system react to "brute-force" denial of service attacks?
- Partially correct protocol runs: messages follow protocol partially correctly

Mutation Operators: Special Characters

*New mutation operators that suit your test objectives can be defined

- Use characters that have special semantics to an interpreter
  - Quotes (‘ ”)
  - Slashes (\)
  - Meta-characters: e.g. single quote, backslash, carriage returns/linefeeds, etc.
  - Escaped data: for double escaping problems
  - HTML: e.g. <b>…
  - Script: e.g. <script>…
  - Others like character encodings

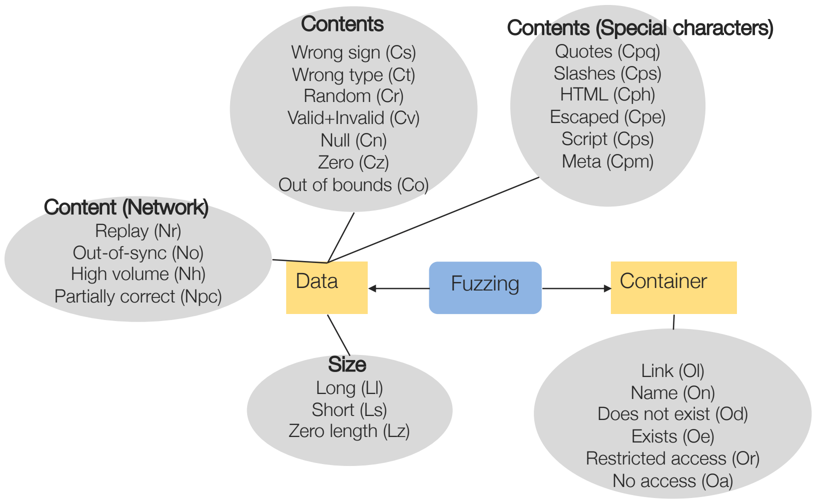
Threat types?

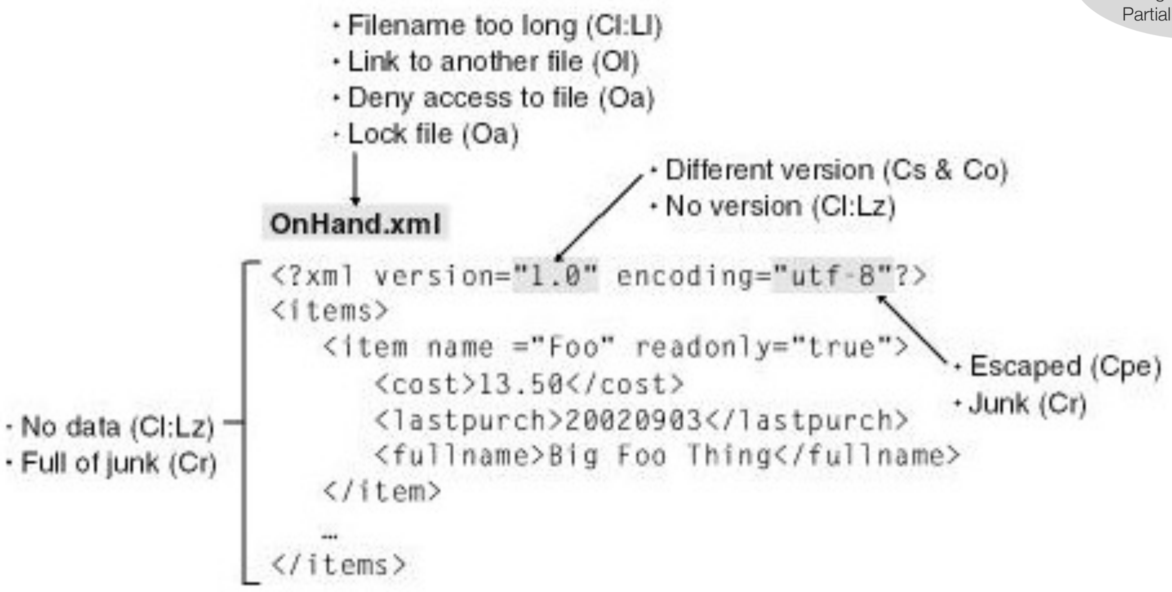


Meta-Characters

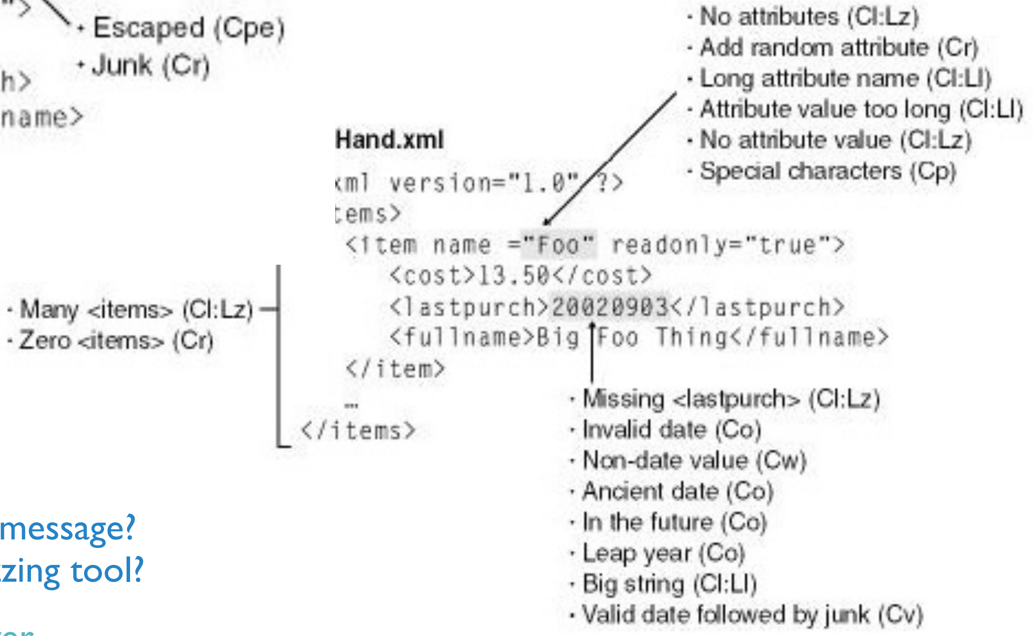
Character	Comments
‘ ‘’	String delimiters
\	Backslash (escape character)
( ) [ ] { }	brackets
<!– and -->	HTML and XML comment operators
--	SQL comment operator
\n and \r or 0x0a and 0x0d	Newline and carriage return
\t	Tab
0x04	End of file
0x7f	Delete
0x00	Null bytes
< and >	Tag delimiters
* ? _	Wildcards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

Example: Mutating XML Data




· Filename too long (Cl:Ll)
· Link to another file (Ol)
· Deny access to file (Oa)
· Lock file (Oa)
OnHand.xml
· Different version (Cs & Co)
· No version (Cl:Lz)
· Escaped (Cpe)
· Junk (Cr)
· No data (Cl:Lz)
· Full of junk (Cr)

```
<?xml version="1.0" encoding="utf-8"?>
<items>
   <item name ="Foo" readonly="true">
      <cost>13.50</cost>
      <lastpurch>20020903</lastpurch>
      <fullname>Big Foo Thing</fullname>
   </item>
   ...
</items>
```


· No attributes (Cl:Lz)
· Add random attribute (Cr)
· Long attribute name (Cl:Ll)
· Attribute value too long (Cl:Ll)
· No attribute value (Cl:Lz)
· Special characters (Cp)
Hand.xml
· Many <items> (Cl:Lz)
· Zero <items> (Cr)
· Missing <lastpurch> (Cl:Lz)
· Invalid date (Co)
· Non-date value (Cw)
· Ancient date (Co)
· In the future (Co)
· Leap year (Co)
· Big string (Cl:Ll)
· Valid date followed by junk (Cv)

```
xml version="1.0" ?>
tems>
 <item name ="Foo" readonly="true">
    <cost>13.50</cost>
    <lastpurch>20020903</lastpurch>
    <fullname>Big Foo Thing</fullname>
 </item>
 ...
</items>
```

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

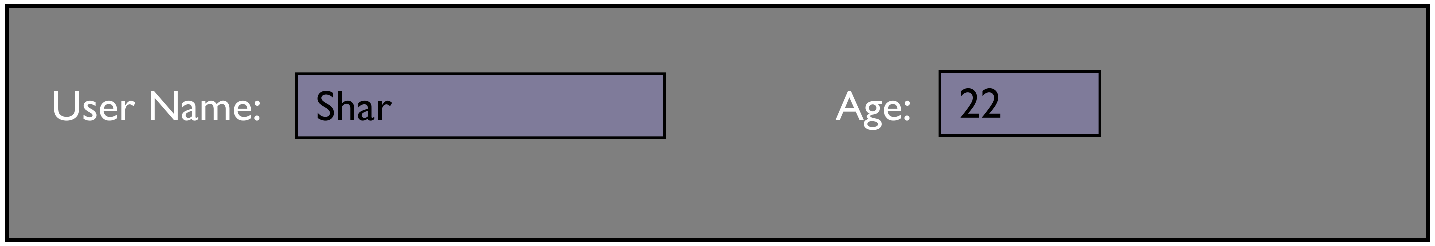
Suggestion: use pair-wise combinations

Example: Mutating Web Form Inputs

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

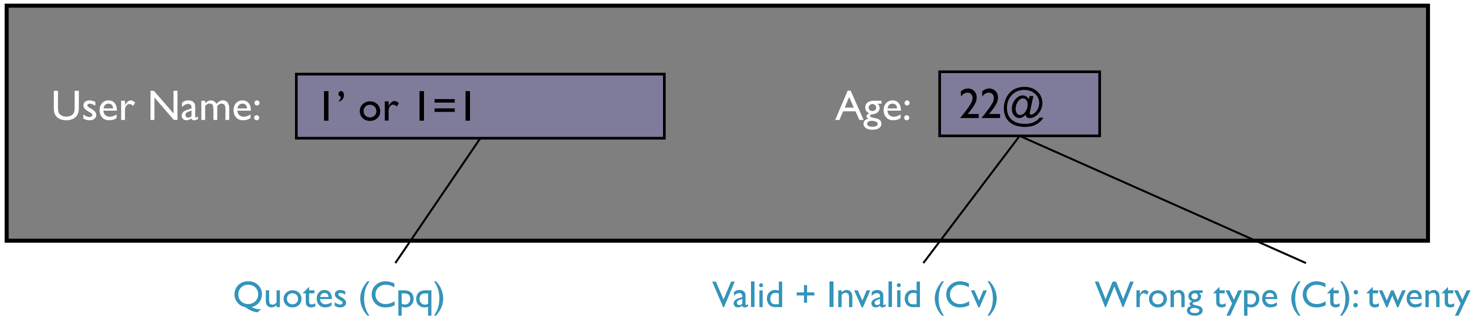
Test Plan:

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



- Mutated Web Form Inputs:

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



Test Oracle: Classifying Output Responses

- **(V) Valid Responses (Expected Outputs):** malformed inputs are adequately processed by the server

(V1)	**Server acknowledges the invalid request and provides an explicit message regarding the violation**
(V2)	**Server produces a generic error message (e.g. HTTP error codes – 401 access defined, 403 Forbidden, 404 Not found)**
(V3)	**Server apparently ignores the request completely**

- **(F) Faults & Failures:** malformed inputs cause abnormal software behaviors, when the code behind the interface being tested fails to handle the input

Randomization?

- How random should random inputs be?
- Random inputs: test how the component handles unexpected inputs; usually not much code exercised as such inputs will be caught by simple input checks
  - When code crashes, check whether the return address has been overwritten by input data → buffer overrun
- Valid + Invalid Operator: use partially incorrect inputs to try to get past first line of defence and exercise more of the code

Domain Knowledge

- You want to mutate data and yet you still want the application to exercise some path properly
- Use the domain knowledge, i.e., start mutating from the valid (structure of) data that you know rather than randomly
- For example, if a web application requires a specific header type, `TIMESTAMP`, setting the data expected in the header to random data is of some use, but it will not exercise the `TIMESTAMP` code path greatly if the code checks for certain values in the header data:
  - This will not be very useful: `TIMESTAMP: H7ahbsk(0kaaR`
  - This may bypass some checks: `TIMESTAMP: 12042018abc`

Practical Problems with (Automated) Fuzzing

- Fuzzing can only detect simple faults or vulnerabilities
- To be effective, it requires significant running time, as it is not efficient
- Fuzzing usually achieves poor code coverage
- Fuzzing typically does not find logic flaws
    - Malformed data likely to lead to crashes, not logic flaws
    - e.g., Missing authentication / authorization checks
- Fuzzing is harder to apply to types of vulnerabilities that do not cause program crashes

Example: Poor Code Coverage

```
<?php
    $action = $_GET[“action”];
    $data = $_GET[“data”];
    if($action==“submit”) {
      record($data);  //sensitive code to test
    }
?>
```

- Setting the `action` parameter to a malformed value doesn’t help much
- Setting `action` to “`submit`” and `data` to a malformed value is a better test

Symbolic Execution

White-box technique

Symbolic Execution

- Analysis of programs by tracking symbolic rather than actual values (static analysis)
- Is used to reason about all the inputs that take the same path through a program
- Builds predicates that characterize
  - Conditions for executing paths: e.g., "`x = 94389`"
  - Effects of the execution on program state: e.g., `ERROR`

```
void test_me(int x) {
  if (x == 94389) {
    ERROR;
  }
}
```

Random Testing

- Generate random inputs and execute the program on those (concrete) inputs
- Probability of reaching `ERROR` could be small: $1/2^{32} = 0.000000023\%$

Predicate

- Predicate: Boolean-valued statement that may be true or false depending on the values of its variables

- **`{x|p(x)} = {x|x equals to 94389}`**

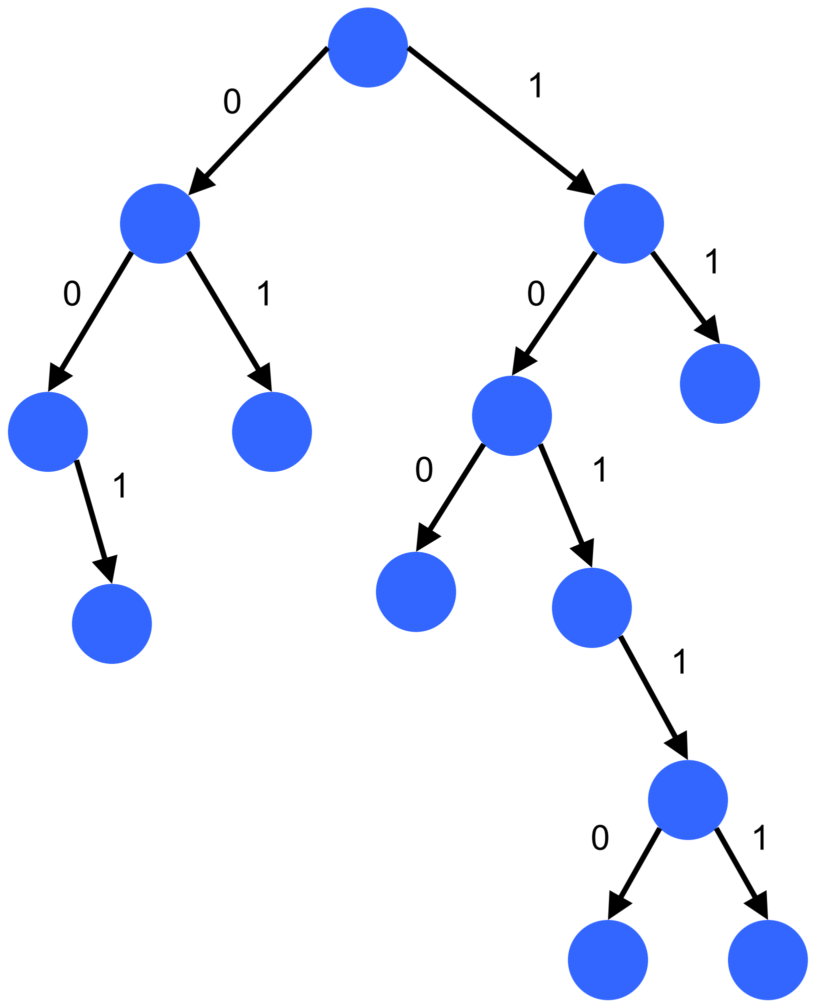
```
p(94389)     T
p(6)         F
```

- **`{x|q(x)} = {x|x is a positive integer less than 5}`**

```
q(2)     T
q(6)     F
```

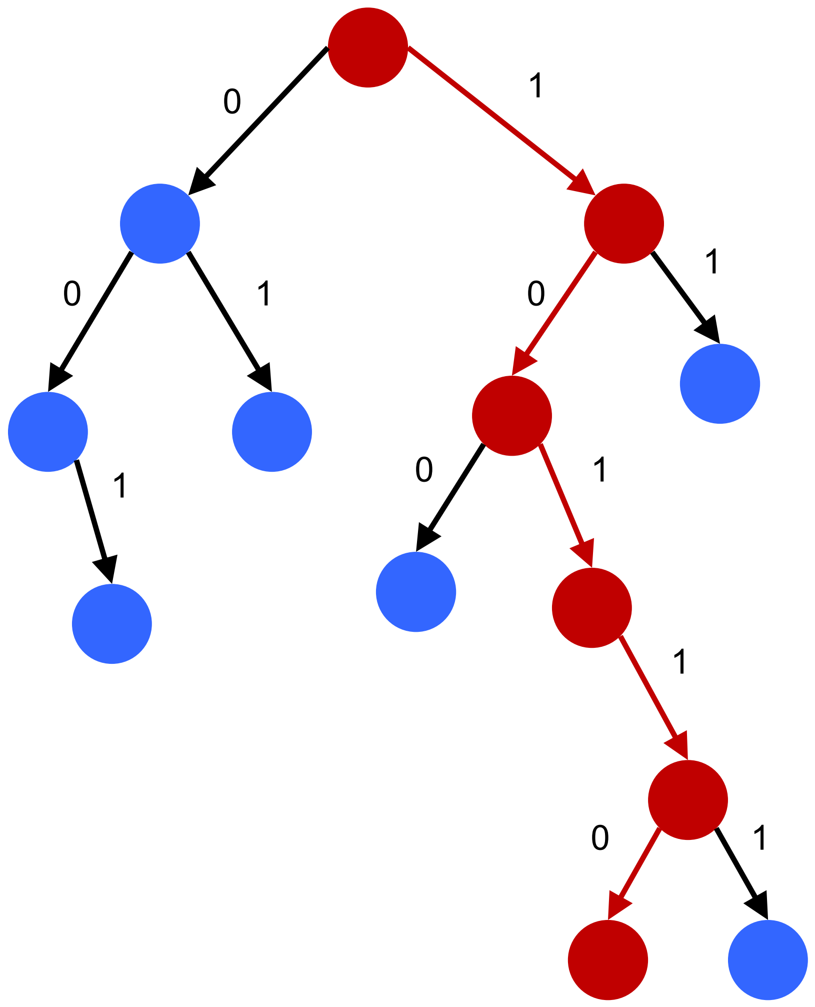
Execution Paths of a Program

- Execution paths of a program can be seen as a binary tree with possibly infinite depth
  - Computation tree
- Each node represents the execution of an if-then-else statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



Execution Paths of a Program

- Execution paths of a program can be seen as a binary tree with possibly infinite depth
  - Computation tree
- Each node represents the execution of an if-then-else statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



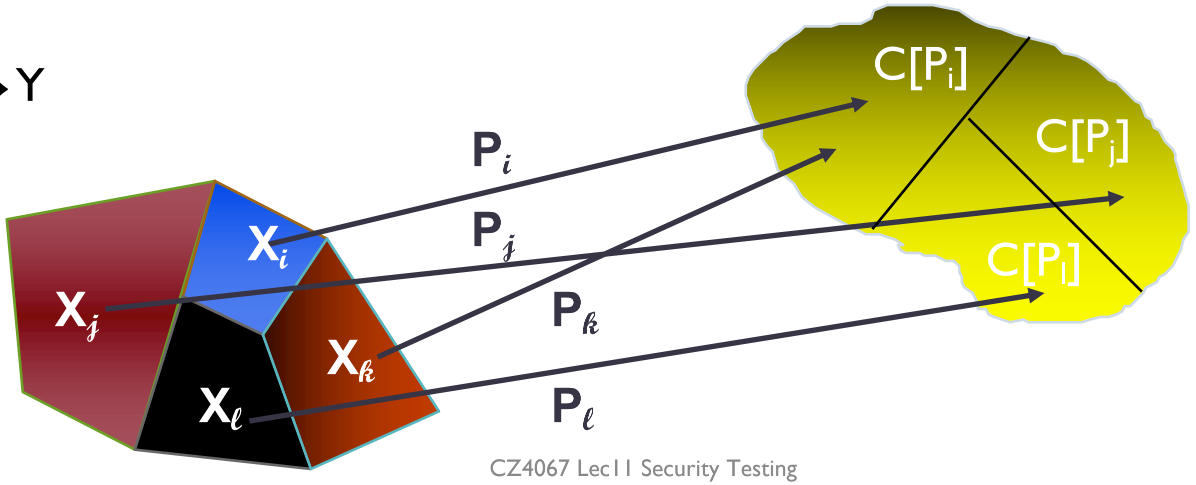
Symbolic Execution

- Uses symbolic values for input variables
- Builds predicates that characterize the conditions under which execution paths can be taken
- Collects symbolic path constraints
- Uses theorem prover (constraint solver) to check if an answer exists and the branch can be taken
- Negates path constraints to take the other side of the condition (branch)

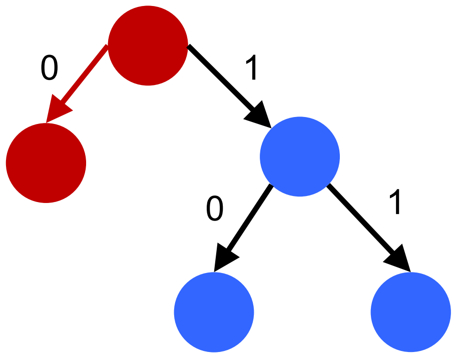
Representing Computation Symbolically

- Symbolic names (X_i) represent the input values
- The path value (PV) of a variable for a path describes the value of that variable in terms of those symbolic names
- The computation of the path $C[P_i]$ is described by the path values of the outputs for the path
- The path condition (PC) describes the domain of the path, and is the conjunction of the interpreted branch conditions
- The domain of the path (D[P]) is the set of input values that satisfy the PC for the path

$P : X \rightarrow Y$



Example program



```
1: int x, y;
2: if(x>y){
3:    x = x+y;
4:    y = x-y;
5:    x = x-y;
6:    if(x-y>0)
7:       assert(false);
8: }
9: return x;
```

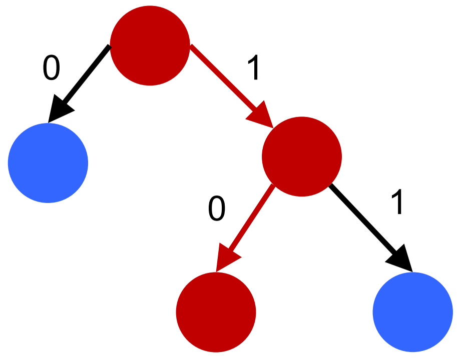
Stmt	PV	PC
1	$x \leftarrow X$ $y \leftarrow Y$	true
2,9	$x \leftarrow X$ $y \leftarrow Y$	$\text{true} \wedge !(X>Y) =$ $X \le Y$

P = 1,2,9

PC = $X \le Y$ D[P] = { $(X,Y) \mid X \le Y$ }

C[P] = PV.x = X

Example program



```
1: int x, y;
2: if(x>y){
3:    x = x+y;
4:    y = x-y;
5:    x = x-y;
6:    if(x-y>0)
7:       assert(false);
8: }
9: return x;
```

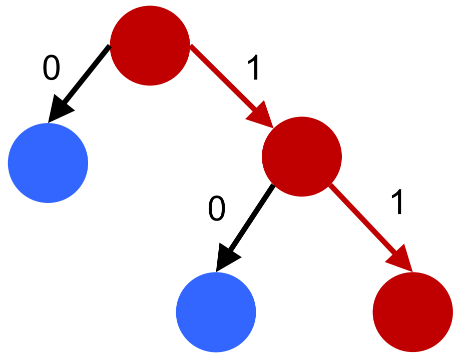
Stmt	**PV**	**PC**
1	$x \leftarrow X$ $y \leftarrow Y$	true
2--5	$x \leftarrow X+Y$ $y \leftarrow (X+Y)-Y=X$ $x \leftarrow (X+Y)-X=Y$	$\text{true} \wedge (X>Y) =$ $X>Y$
6,9	$x \leftarrow Y$ $y \leftarrow X$	$X>Y \wedge !(Y-X>0) =$ $X>Y$

P = 1,2,3,4,5,6,9

PC = X>Y

C[P] = PV.x = Y

Example program



Error is not reachable

```
1: int x, y;
2: if(x>y){
3:    x = x+y;
4:    y = x-y;
5:    x = x-y;
6:    if(x-y>0)
7:       assert(false);
8: }
9: return x;
```

Stmt	**PV**	**PC**
1	$x \leftarrow X$ $y \leftarrow Y$	true
2--5	$x \leftarrow X+Y$ $y \leftarrow (X+Y)-Y=X$ $x \leftarrow (X+Y)-X=Y$	true $\wedge$ (X>Y) = X>Y
6,7	$x \leftarrow Y$ $y \leftarrow X$	X>Y $\wedge$ (Y-X>0) = FALSE!

P = 1,2,3,4,5,6,7

PC = false (Infeasible Path!)

C[P] = ERROR!

Constraint Solving

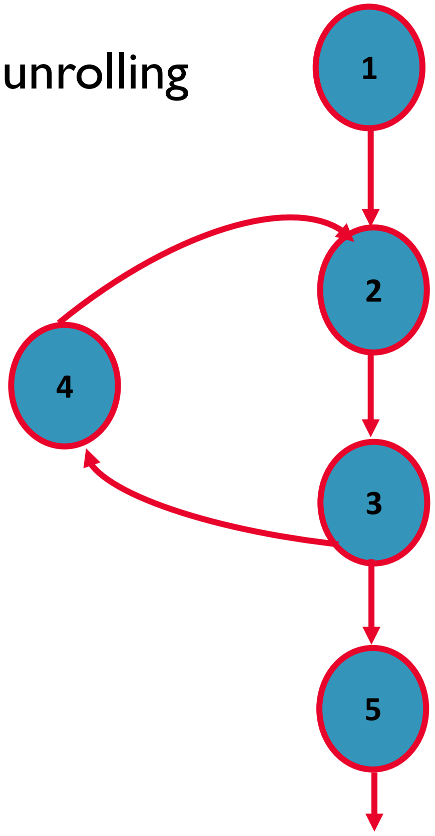
- Mathematically determine if a set of constraints can be solved
- Some modern constraint solvers:
  - Z3, CVC, LP solver

- Primarily linear equations:
  - `if (3 + x < 6) { y = x * 2; } else { … }`
    - Constraint to solve: **`(3 + x < 6)`**
    - What is a value of `x` in order to cover the IF branch?
  - Constraint Solver: `(x < 3) -> x = 2`
  - (IF branch covered)

  - Negate constraint: **`(3 + x >= 6)`**
  - Constraint Solver: `x >= 3 -> x = 4`
  - (Else branch covered)

Handling Loops – our options

- Use loop unrolling



Example Paths

- P= 1, 2, 3, 5
- P= 1, 2, 3, 4, 2, 3, 5
- P= 1, 2, 3, 4, 2, 3, 4, 2, 3, 5
- Etc.

- Create and prove correct loop invariants

Problems with Symbolic Execution

- Expensive to create program representations
- Expensive to reason about expressions
  - Although modern SAT solvers help!
- Problems with function calls – need to keep track of calling contexts
  - Called inter-procedural analysis
- Problem with handling loops
  - often unroll them up to a certain depth rather than dealing with termination or loop invariants
- Aliasing – leads to a massive blow-up in the number of paths

Symbolic Execution Engines

- Symbolic Java PathFinder
- KLEE (for C)
- Cloud9 — parallel symbolic execution, also supports threads
- Pex — symbolic execution for .NET
- jCUTE — symbolic execution for Java
- Otter — symbolic executor for C
- RubyX — symbolic executor for Ruby

Test Selection / Prioritization

Why Test Prioritization?

Problems

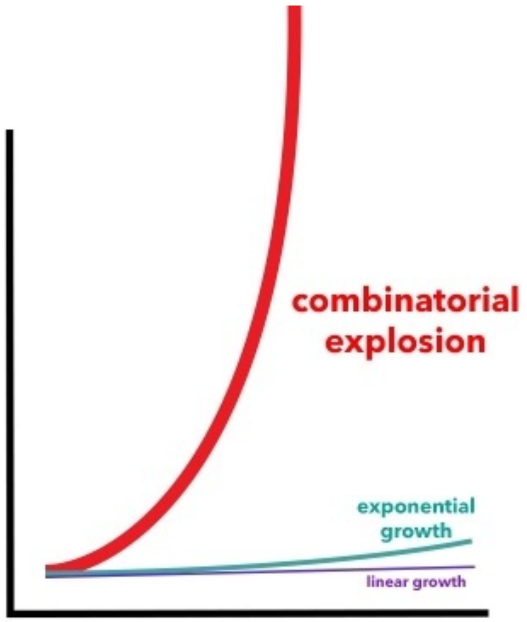
- Huge number of test cases
- High testing cost
- In practice budget and time constraints pose barriers to the testing process

Proposals

- Test in a best effort basis
- Maximize the test effectiveness per utilized test
- Stop the testing process when the constraints are met (manpower, budget, deadline)
- Consider removing the features that were not tested or at least disable them by default

Combinatorial Testing

- A system under test (SUT) may have many input parameters, each with many possible values
- To (security) test a system properly, we need to consider the interactions between parameters
- However, testing for all possible combinations of those parameters with many possible values is impossible – combinatorial explosion problem
  - E.g., 4 input parameters, each with 6 possible values
- We therefore need to cope with combinatorial explosion; find input spaces that are more likely to lead to (security) bugs
  - Pareto's Law: approximately 80% of bugs come from 20% of modules

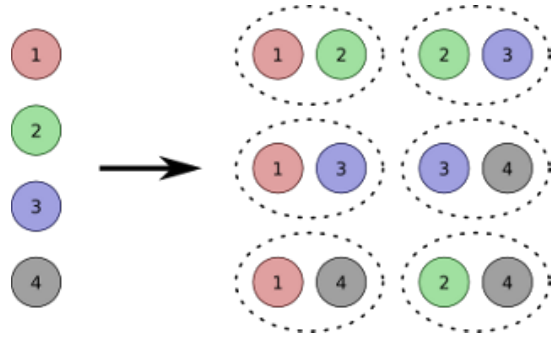


→ 1296 test cases

http://www.testrisk.com/2013/11/combinatorial-testing.html

Pairwise Testing

- Combining input parameters in a systematic way
- Testing all combinations takes a long time; pair-wise testing is very efficient for finding defects
- Test at least one combination for every combination of each two-parameters



Rationale:

Most common bugs are generally triggered by either a single input parameter or an interaction between pairs of parameters

Bugs involving interactions between three or more parameters are less common and more expensive to find

Cost-Benefit compromise

Example: Pairwise Testing

Parameters

Name: Jack, John, Mike
Role: User, Admin
Action: Create, Update

All Combinations (3*2*2 = 12)

	Name	Role	Action
TC1	Jack	User	Create
TC2	Jack	User	Update
TC3	Jack	Admin	Create
TC4	Jack	Admin	Update
TC5	John	User	Create
TC6	John	User	Update
TC7	John	Admin	Create
TC8	John	Admin	Update
TC9	Mike	User	Create
TC10	Mike	User	Update
TC11	Mike	Admin	Create
TC12	Mike	Admin	Update

Pair-wise Combinations (3*2=6)

	Name	Role	Action
TC1	Jack	User	Create
TC4	Jack	Admin	Update
TC6	John	User	Update
TC7	John	Admin	Create
TC9	Mike	User	Create
TC12	Mike	Admin	Update

A test-suite generated by Pair-wise Combinations is not unique!

Tool support:
https://github.com/Microsoft/pict

Example: Fuzzing + Pair-wise Combinations

Input Parameters

Name: Jack, John, Mike
Role: User, Admin
Action: Create, Update

Mutation Operators

Operator 1: Null
Operator 2: Valid+Invalid

Pair-wise Combinations

	Name	Role	Action
TC1			
TC2	Jack?		Create!
TC3		User1	Update?
TC4	John!	Admin2	

Quiz

- I have four parameters (a, b, c, d) that have (2, 10, 5, 3) possible values respectively: 2 values for a, 10 for b, etc.
  a) What is the total number of test cases for testing all possible interactions among parameters?
  b) What is the minimum number of test cases if pair-wise method is used?

(a) 2*10*5*3 = 300

(b) 10*5 = 50

Security Testing In Action

Yet another way to earn some money in this course …

Scantist Bug Bounty Program

SCANTIST

Submit a vulnerable repository to us and the winner of each language will receive **$10 Grab voucher** or **$10 Starbucks voucher** from us!

How to do it?

1. Please attend the **tutorial next week (6 April)** to know more about it!

How to win it?

1. For every language we supported (page 4), you can submit a vulnerable repository with the repo URL
2. In each language, we will pick a repo that has the highest number of stars and contain at least 1 high/critical vulnerability. If there are multiple repos that have the same number of stars, then the repo that has the highest score vulnerability will be picked. If it is still a close fight, then the repo that has the most number of high/critical vulnerability will be selected
3. Once the winning repo of each language is selected, the first person to submit will be the winner
4. One person can submit multiple times for a language
5. One person can submit on multiple languages

Conclusions

Summary

- Secure software must be able to handle intentionally malformed inputs
- Your code therefore has to detect malformed inputs. Don't trust your inputs!
- Test your code to detect whether there are malformed inputs that are not detected
- For the malformed inputs detected, pay attention to the error handlers

Conclusion

- Discussed the role of the security tester and how your job is not to prove that features work; rather, it is to determine how you can make features work in ways not anticipated or intended
- Use the threat model to determine the components within the application that require test plans
- Use the STRIDE threat categories to decide what techniques to use to test that the threat is mitigated
- Data mutation is an incredibly useful way to force an application to fail

Reference

- Michael Howard and David LeBlanc, Writing Secure Code, Microsoft Press 2nd Edition, 2003