RUN TIME PROTECTION

Lect 10

Run-time hardening

- It is important to protect the run-time in an OS. For Linux systems this is very important as default configurations in a distribution leave too many functionality open for misuse

- There is a lot information how to do this and some distributions have guidelines for hardening

- 

Red Hat Enterprise Linux 7

Security Guide

A Guide to Securing Red Hat Enterprise Linux 7

Runtime protection

- Many operating system deploy forms of protection to make it harder for malware to be successful
  - E.g address layout randomization, stack canaries, etc.
  - Remember also IMA
- A popular and useful protection is the non-execute flag for data in memory. It is supported via HW features in ARM, AMD and Intel HW.
- (and used in Android and iOS, etc)
- BUT

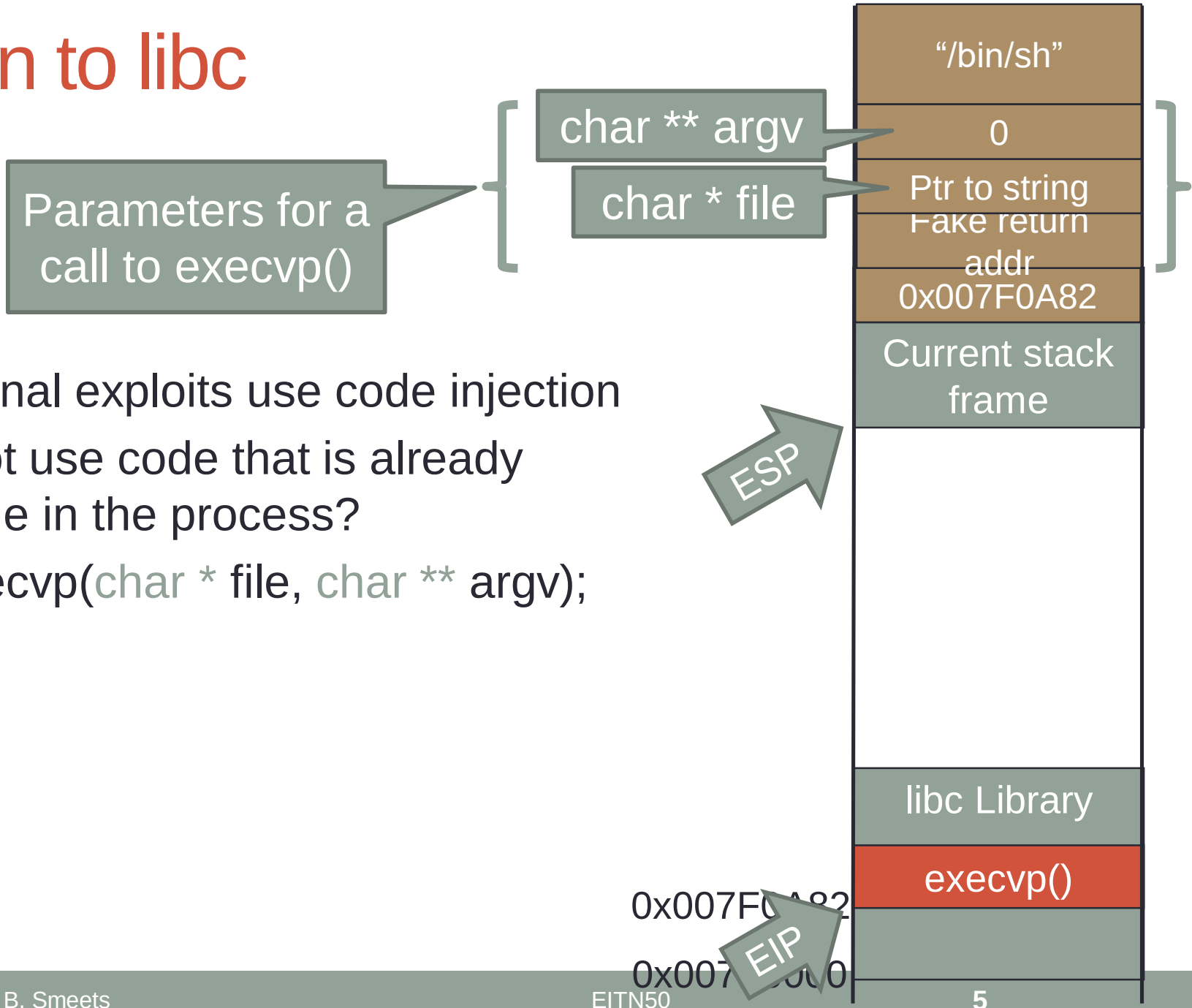
When NX bits Fail

In many CPU one can mark a page as executable

- AMD, ARM calls it the NX bit: No-eXecute
- Intel calls it the XD bit: eXecute Disable

- NX prevents shellcode from being placed on the stack
  - NX must be enabled by the process
  - NX must be supported by the OS
- Can exploit writers get around NX?
  - Of course ;)
  - Return-to-libc
  - Return-oriented programming (ROP)

Return to libc



- Traditional exploits use code injection
- Why not use code that is already available in the process?

execvp(char * file, char ** argv);

Stack Control = Program Control

- Return to libc works by crafting special stack frames and using existing library code
  - No need to inject code, just data onto the stack
- Return-Oriented Programming (ROP) is a generalization of return to libc
  - Why only jump to existing functions?
  - You can jump to code **anywhere** in the program
  - Gadgets are snippets of assembly that form a Turing complete language
  - Gadgets + control of the stack = arbitrary code execution power

ProjectD

BLOCKCHAINS

bitcoin

HYPERLEDGER

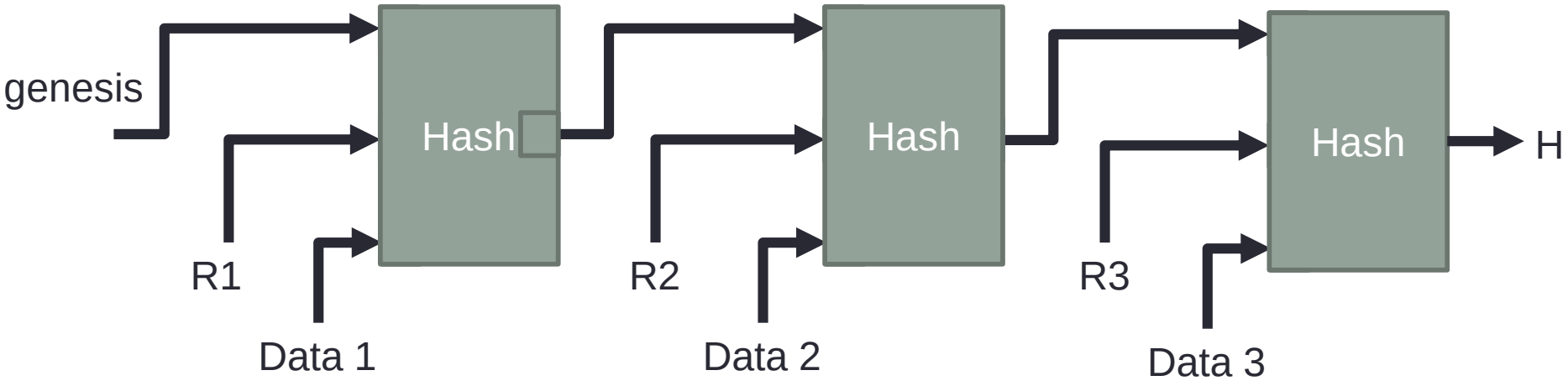
Outline

Key concepts

- Merkle trees
- hash chains
- smart contract
- consensus
- proof-of-work

Public vs Permissioned blockchains

Hash chains

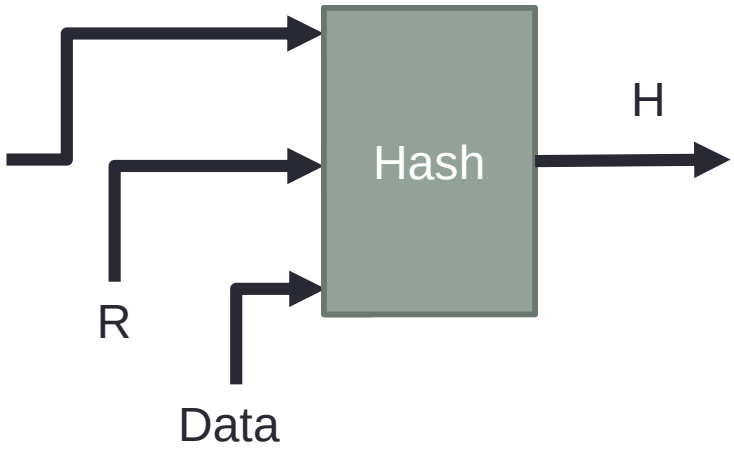


Output depends on the genesis (start), all data inputs, and all the R values

R's can be counters or values that make each hash value to have a prescribed property, e.g. first 10 bits of output should be zero.

Proof-of-work

The **effort** to obtain values R such that the out hash has a given n-bit pattern, given the previous hash and data (hash), is



2^n trials

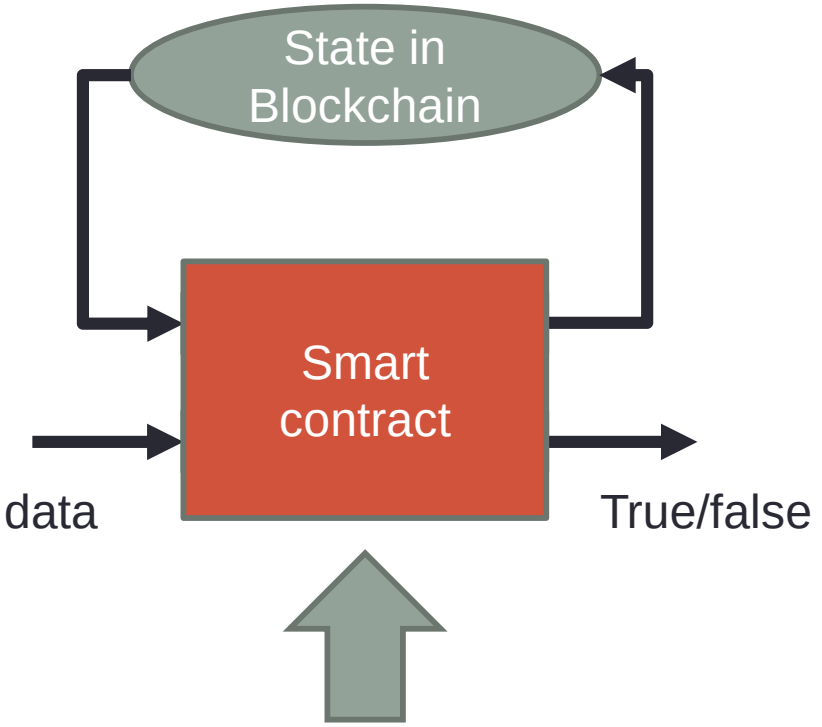
H= XXXXXXXXXXX0000000

n-bit pattern

e.g. in Bitcoin it takes on average 10 minutes to do the search (= mining effort)

Smart contract

Instead of using directly the data as input to the hash chain one takes the input to a program that is stored in the blockchain, this program is called a smart contract and executed with the data, and optionally a state, it will return true/false and a possible updated state



Language: e.g. Ethereum uses the language Solidity

Example Solidity contract

```
pragma solidity ^0.4.11;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    function WithdrawalContract() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() payable returns (bool) {
        if (msg.value > mostSent) {
            pendingWithdrawals[richest] += msg.value;
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }

    function withdraw() {
        uint amount = pendingWithdrawals[msg.sender];
        // Remember to zero the pending refund before
        // sending to prevent re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}
```

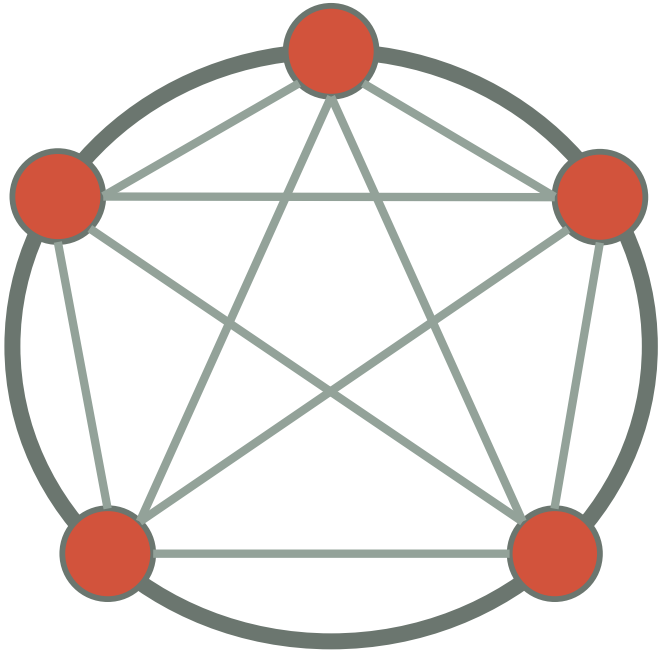
https://solidity.readthedocs.io/en/develop/

Consensus

Before the blockchain data is updates the new entries are “reviewed” by a set of nodes. The nodes have to arrive to a consensus before the blockchain is updated.

In practice we want to handle consensus in the presence of malfunction or dishonest nodes.

Nodes interact, e.g. using the Paxos algorithm, to come to consensus



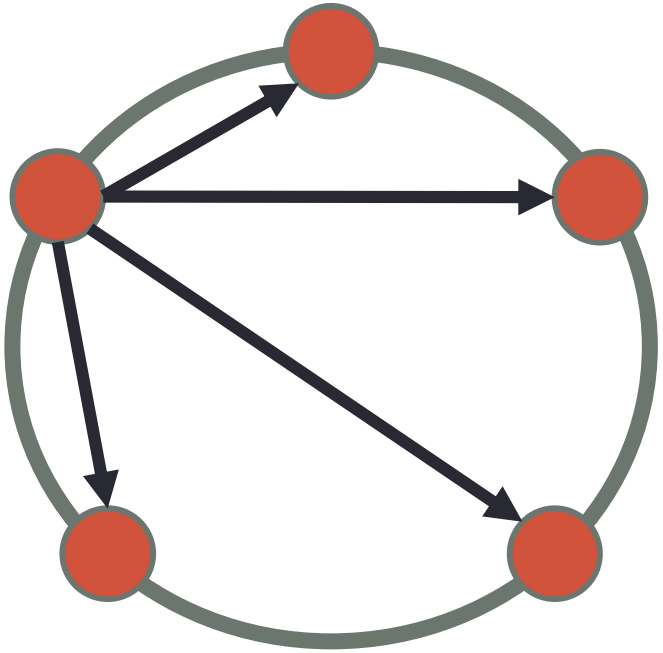
“the Byzantine general problem”

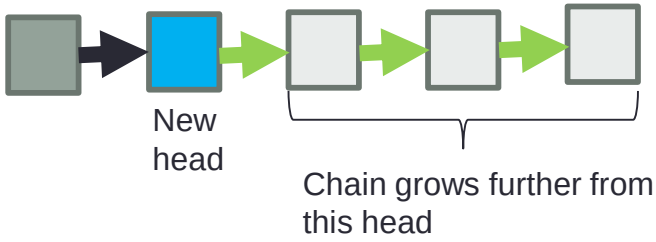
Consensus in Bitcoin

In Bitcoin the node (miner) that succeeds in the proof-of-work announces this to other miners and that check and confirm this and extend the blockchain from this new blockchain head.

Consensus is effectively established if the blockchain is consistently growing from the new head for several extension of the chain

Winning miner announces his/her win to other miners





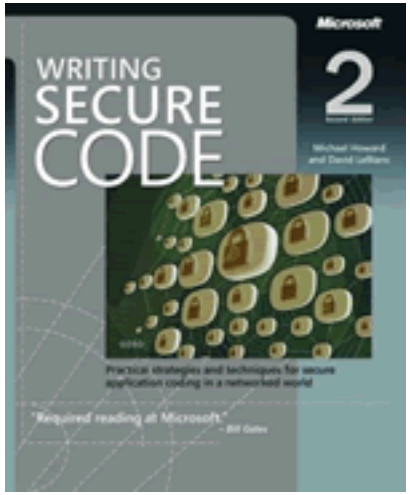
Permission vs Public blockchains

In a public blockchain anyone can be a “miner” that is involved the work to extend the blockchain.

- Proof-of-work is essential
- Built-in reward system needed to incentivize people to become a miner

- Public blockchains must be slow
- Consensus in general makes blockchains slow
- Permission blockchain require scheme for access authentication.
- How to distribute genesis securely??

SOFTWARE/CODE SECURITY



Outline

- Two sides of software security
  - Secure applications
- Protecting software
- Secure software development
  - Assurance: formal verification, proofs

Two sides of software security

When people speak of software security they can mean

- **Ways to protect the use of software**
  - Licensed software
  - Software that others might want to steal/reuse

OR

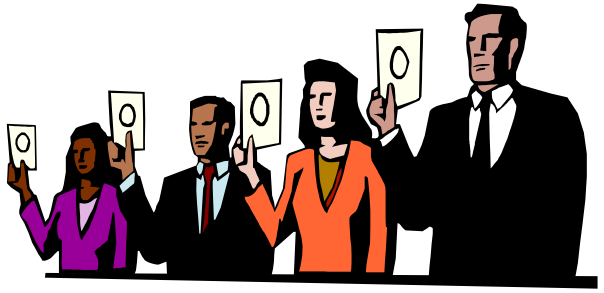
- **Ways to write high quality software for mission critical applications**
  - Free of bugs
  - Well-defined in its behaviour

Secure Applications

By *secure* we mean here that the apps resists attack from common threat vectors, and secure apps minimize the number entrances for attack

- Secure apps minimize the number of failure modes
- Secure apps are difficult to reverse engineer
- Secure client apps may have some copy protection

Definitions;
An attack/threat vector is a path or means by which a hacker can perform his/her attack.



The collection of attack vectors forms the attack surface.

PROTECTING SOFTWARE IP

Protecting SW IP: Outline

SW protection is becoming increasingly important. This session will discuss the various software security mechanisms

Topics covered include

- license enforcement,
- obfuscation,
- encryption, and
- authentication.

Why do we want software protection?

- Writing good software is expensive
  - Secure return of investment
  - Prevent competitors to get quickly onpar

- Using the software is coupled to ways of earning money
  - Service
  - Selling
  - Licensing
  - …

Forms of Software Piracy 

Usage type restrictions for software

- Only (paying) customers should be able to use the SW
  - Licensed customers
    - Network Based License
    - Personal license
    - Limiting concurrent executions
  - Other users
    - Trial “Time Out” Version

Common Approaches to Software copy protection

- Anti-Debug routines
- Additional Hardware requirement (Dongle)
- Unique keys
- Encryption or Obfuscation of Application components
- Modification checking algorithms
- Provide only as a service, e.g. as cloud service

Solutions to limit use of local SW copies

- Hardware Dongles
  - Pro: Possibly more robust security
  - Con: Inconvenient, distribution logistics cost/problems

- Software Only
  - Provide copy protection and licensing via keying schemes and license servers

Dongles often contain an authentication mechanism that the allows the SW code to do a challenge response to check if a license is present

Things to consider with licensing sw copies

- Must balance apps cost and value
- Risk of Compromise
- Support Strategy
- Customer Needs
- Distribution Mechanism

Still used, e.g. VHDL tools.

In the future we may see shift to Cloud services

Using Java or .NET?
- how the code looks like

- Recall: Java and .NET Framework programs compile to *intermediate* languages
- Under native compilation, symbols are left out
- Not so with Java or.NET Framework Apps!
  - Free decompilers exist to recreate source code from compiled programs (e.g. JD Java Decompiler)
- Similar problems with Android and iOS apps

Reverse Engineering

- Process of backtracking through the software process
- Obtaining source code from binary/byte code.
- Understanding programs to realize intent.
- Intellectual property issues.

Project 5: is partly related to this

The Problem with reverse engineering

- Trade secrets exposed
- Security holes exposed
- Intellectual Property lost
- Easy to crack programs
- Competitors can steal your product

Why not just Encrypt SW?

Encryption is like putting your application in a lockbox

- To execute, the runtime needs the key

- So, if the runtime can get it, so can crackers

SW as Service

- For example: the compiler is not delivered as a program to the user but the user can go to a (cloud) service to have his/her program compiled.

- **Pros**: SW is protected behind the server out of reach of the user.

- **Cons:** Now we have an access problem. How to give only legitimate (paying customers) access to the service.

  This might be easier but is a cost factor.

What is Obfuscation?

- Obfuscate – “to confuse”

  Transform the code flow/appearance
- so the code becomes hard to trace and understand
- Functional runtime behavior is the same (not necessary memory and time costs)

Source code Obfuscation – (simple)Perl example

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU / lreP
  rehtona tsuJ";sub p{
  @p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=$f=!
  fork;map{$P=$P[$f^ord ($p{$_})&6];$p{$_}=/
  ^$P/ix?$P:close$_}keys%p}p;p;p;p;p;map{$p{$_}=~/^[P.]/
  && close$_}%p;wait
  until$?;map{/^r/&&<$_>}%p;$_=$d[$q];sleep
  rand(2)if/\S/;print
```

A classification of obfuscations

- Layout transformations
  - Change formatting information
- Control transformations
  - Alter program control and computation
- Aggregation transformations
  - Refactor program using aggregation methods
- Data transformations
  - Storage and encoding information

Deobfuscation

- Almost all obfuscating transforms have a deobfuscating transform
  - Program slicing
  - Pattern matching
  - Statistical analysis
  - Data flow analysis
  - Theorem proving

Code Obfuscation – does it really work?

- **Experience** with Java (and .NET) shows that obfuscators do give protection – to some extend

- For C/C++ there exist tools that have reported to be useful, e.g. ActiveCloak, Arxan (commerical)

- **Theoretically**: there is no hope that it works (in general)

  See: On the (Im)possibility of Software Obfuscation,

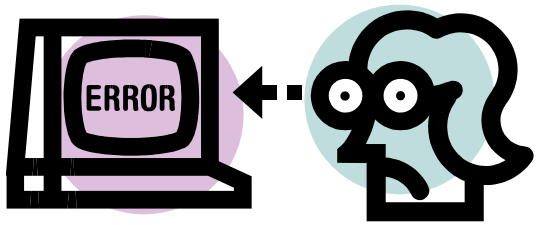
  www.math.ias.edu/~boaz/Papers/obfuscate.pps

  But under relaxed conditions the odds are better:

  new (2014) are of research

SECURE SOFTWARE DEVELOPMENT

Material from 2nd Fundamental Practices for Secure Software Development
http://www.safecode.org/



Contents

- General considerations
- Modeling threats
- Good practices for writing secure code
- Proof correctness of programs

Security as a feature or ?

- Important to Developers/Management are:
  - Features
  - Deadlines

- Security is not always seen as a feature

Writing SW and steam engines

- Laws of physics: often they save us from stupidity
- In programming there are no such laws

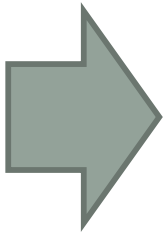
Use Least Privilege

This concept comes from Saltzer and Schroeder's seminal paper, "The Protection of Information in Computer Systems." Proceedings of the IEEE, Sept. 1975

"Every program and every user of the system should operate using the least set of privileges necessary to complete the job."

Overview of good practices

- Secure design principles
  - Secure Information Flow Analysis
- Secure coding practices
- Testing recommendations
  - Static and Dynamic code analysis
  - Taint analysis
- Technology recommendations



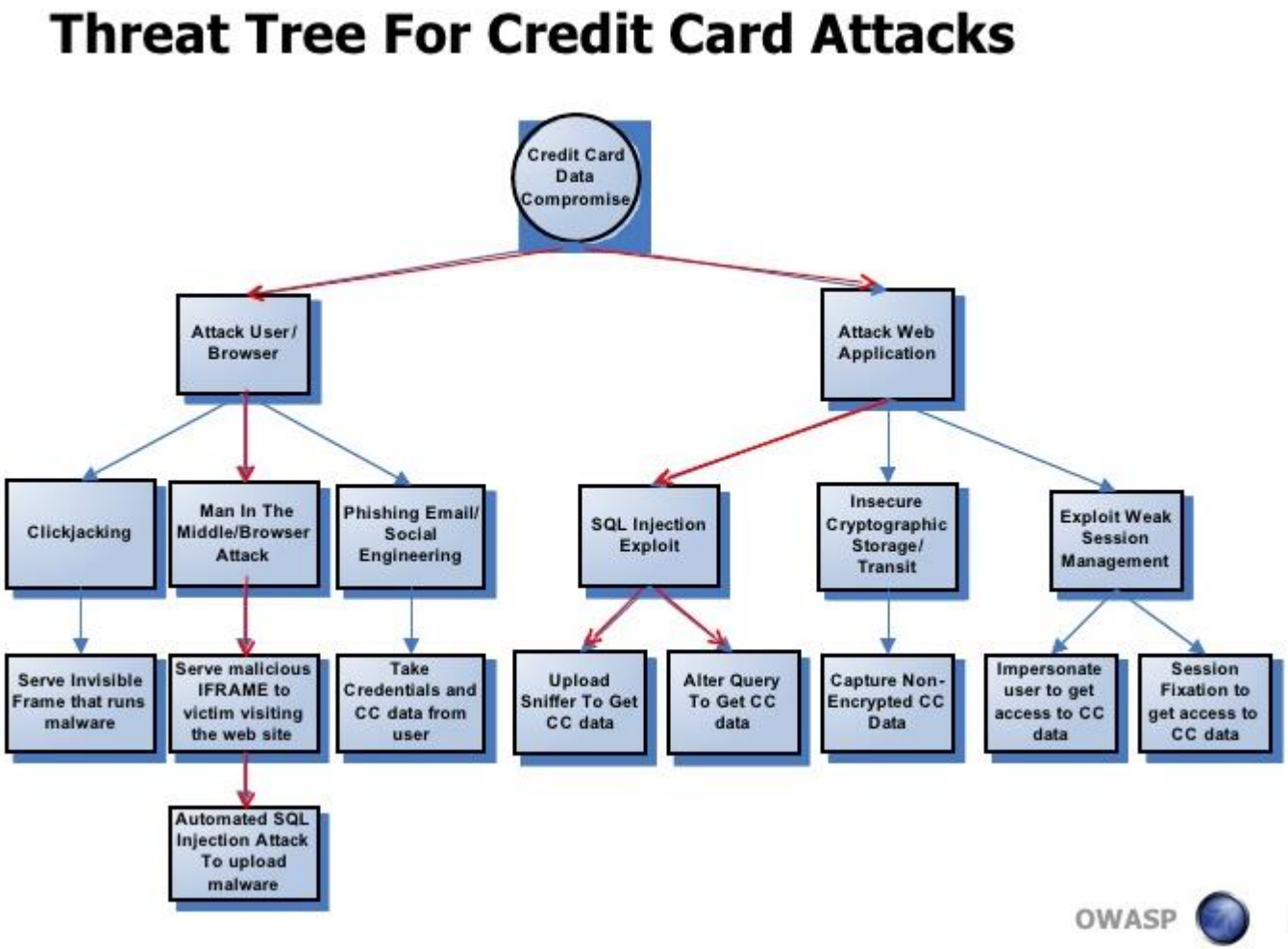
Challenge: all this has to work with Your SW development process

Threat Model

Make a threat model for your application:

- Categorize Threats
  - Spoofing, Tampering, Info Disclosure, Denial of Service, Privilege Elevation
- Build Threat/attack Tree
- Rank Threats
  - Potential Damage, Exploitability, Affected Users, Reproducibility, Discoverability

Attack Threat/Trees



Often used approaches for threat modeling

- **STRIDE**
  - this methodology classifies threats into 6 groups: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege

- **Misuse cases**
  - The employment of misuse cases helps drive the understanding of how attackers might attack a system.

- **“Brainstorming”**
  - if an organization does not have expertise in building threat models, having a security-oriented discussion where designers and architects evaluate the system is better nothing at all.

Tools for threat analysis

Tools that assist in threat analysis do exist but this is still an area where much progress is needed to meet the needs to capture real-life use cases.

Require much experience in use and application to actual cases.

ADTree: tool for attack-defence trees

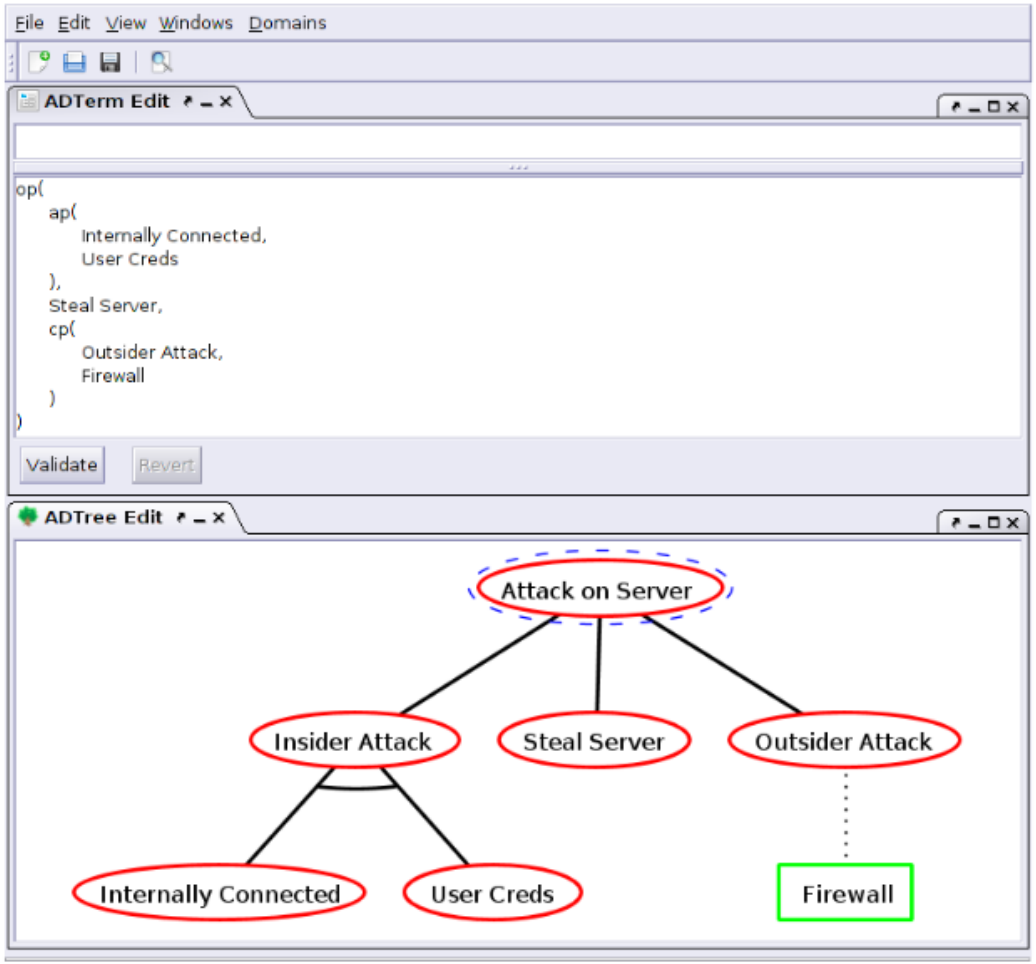


Fig. 2. An ADTree modeled in ADTool.

INFORMATION FLOW ANALYSIS

Collected From:

Mobile device security: Adam C. Champion and Dong Xuan

Problem

How to prevent stored sensitive information from being leaked

Why is this a challenge:

- **Access control** can only prevent information from being **released** not from being **propagated**
- **Trusting programs** is difficult and cannot be static (updates and underying OS or libs change)
- **Monitoring** in- and outputs is not sufficient

Definition

- Static analysis of program to prove that there are no faults and leaks
- If the program passes the analysis then it can be safely executed

Information flow

- Classify program variables into different security levels: $L, H: L \leq H$
- Prevent information flows from higher level variables to lower level ones

- **Example** $secret: H, leak: L$
  - Legal flows

    $secret = leak;$
  - Dangerous flows

    $leak = secret;$

    or

    $if\ ((secret\ \%\ 2) == 0)$

    $\quad leak = 0;$

    $else$

    $\quad leak = 1;$

Recall Bell-Lapadula from your basic computer security course at EIT

How to analyse?

- Taint Analysis
  - Usual through instrumentation of existing code
  - Track information (variables) when being used
  - Many tools: TaintDroid, TaintPhp, TaintCheck

- Secure Information Flow (SIF) Analysis
  - Formal approach: e.g. languages FlowCaml

Secure coding practices

- Minimize Use of Unsafe String and Buffer Functions
- Automatic use of safer functions
  - Visual C++:
    - -D_CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES=1
  - gcc:
    - –D_FORTIFY_SOURCE=2 –O2
- Validate Input and Output to Mitigate Common Vulnerabilities

Unsafe Function	Safer Function
strcpy	strcpy_s
strncpy	strncpy_s
strcat	strcat_s
strncat	strncat_s
scanf	scanf_s
sprintf	sprintf_s
memcpy	memcpy_s
gets	gets_s

Secure coding practices

- Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets

  Common issues leading to vulnerabilities:
  - Overflow and underflow
  - Signed versus unsigned errors
  - Data truncation
- Use Canonical Data Formats
  - \\127.0.0.1\C$\my files\Hello World.docx
- Use Anti-Cross Site Scripting (XSS) Libraries

STATIC & DYNAMIC CODE ANALYSIS

From:

Static and dynamic analysis: synergy and duality, Michael Ernst

Static analysis

Examples: compiler optimizations, program verifiers

Examine program text (no execution)

Build a model of program state

- An abstraction of the run-time state

Reason over possible behaviors

- E.g., “run” the program over the abstract state

Abstract interpretation

Typically implemented via dataflow analysis

Each program statement's *transfer function* indicates how it transforms state

Example: What is the transfer function for

```
y = x++;
```

Selecting an abstract domain

⟨ x is odd; y is odd ⟩

```
y = x++;
```

⟨ x is even; y is odd ⟩

⟨ x = { 3, 5, 7 }; y = { 9, 11, 13 } ⟩

```
y = x++;
```

⟨ x = { 4, 6, 8 }; y = { 3, 5, 7 } ⟩

⟨x=3, y=11⟩, ⟨x=5, y=9⟩, ⟨x=7, y=13⟩

```
y = x++;
```

⟨x=4, y=3⟩, ⟨x=6, y=5⟩, ⟨x=8, y=7⟩

⟨ x is prime; y is prime ⟩

```
y = x++;
```

⟨ x is anything; y is prime ⟩

⟨ $x_n = f(a_{n-1},\ldots,z_{n-1})$; $y_n = f(a_{n-1},\ldots,z_{n-1})$ ⟩

```
y = x++;
```

⟨ $x_{n+1} = x_n+1$; $y_{n+1} = x_n$ ⟩

Static analysis recap

- Slow to analyze large models of state, so use abstraction
- Conservative: account for abstracted-away state
- **Sound:** (weak) properties are guaranteed to be true

*Some static analyses are not sound

Challenges:

- Choose good abstraction

  The abstraction determines the expense (in time and space)

  The abstraction determines the accuracy (what information is lost)

  - Less accurate results are poor for applications that require precision
  - Cannot conclude all true properties in the grammar

- False positives (and false negatives)
  - **A false positive** is a code fragment that is erroneously flagged as being dangerous. This may lead to correct code being replaced by code that has an error (and this happens in practice!)
  - **False negatives** where we miss to identify code with a problem is problematic too. Usually one has to find a balance

Dynamic analysis

Examples: profiling, testing

Execute program (over some inputs)

- The compiler provides the semantics

Observe executions

- Requires instrumentation infrastructure

Must choose what to measure, and what test runs

Challenge: What to measure?

- Coverage or frequency
  - Statements, branches, paths, procedure calls, types, method dispatch
- Values computed
  - Parameters, array indices
- Run time, memory usage
- Test oracle results
- Similarities among runs: towards soundness
- Like abstraction, determines what is reported

Challenge: choose good tests

The test suite determines the expense (in time and space)

The test suite determines the **accuracy** (what executions are never seen)

- Less accurate results are poor for applications that require correctness

*What information is being collected also matters

Dynamic analysis recap

- Can be as fast as execution (over a test suite, and allowing for data collection)
  - Example: aliasing
- Precise: no abstraction or approximation
- Unsound: results may not generalize to future executions
  - Describes execution environment or test suite

Static analysis

Abstract domain
  slow if precise

Conservative
  due to abstraction

Sound
  due to conservatism

Dynamic analysis

Concrete execution
  slow if exhaustive

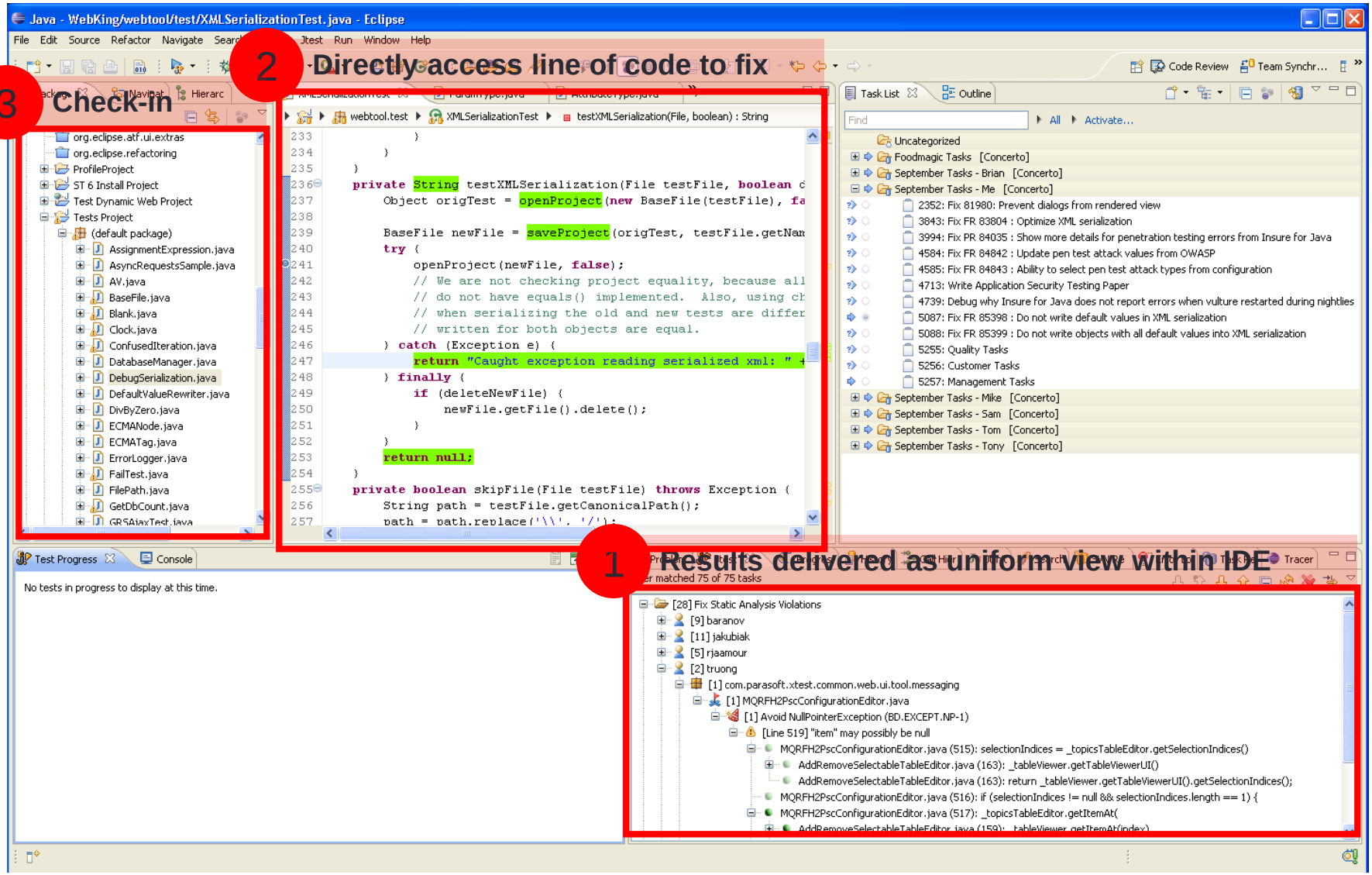
Precise
  no approximation

Unsound
  does not generalize

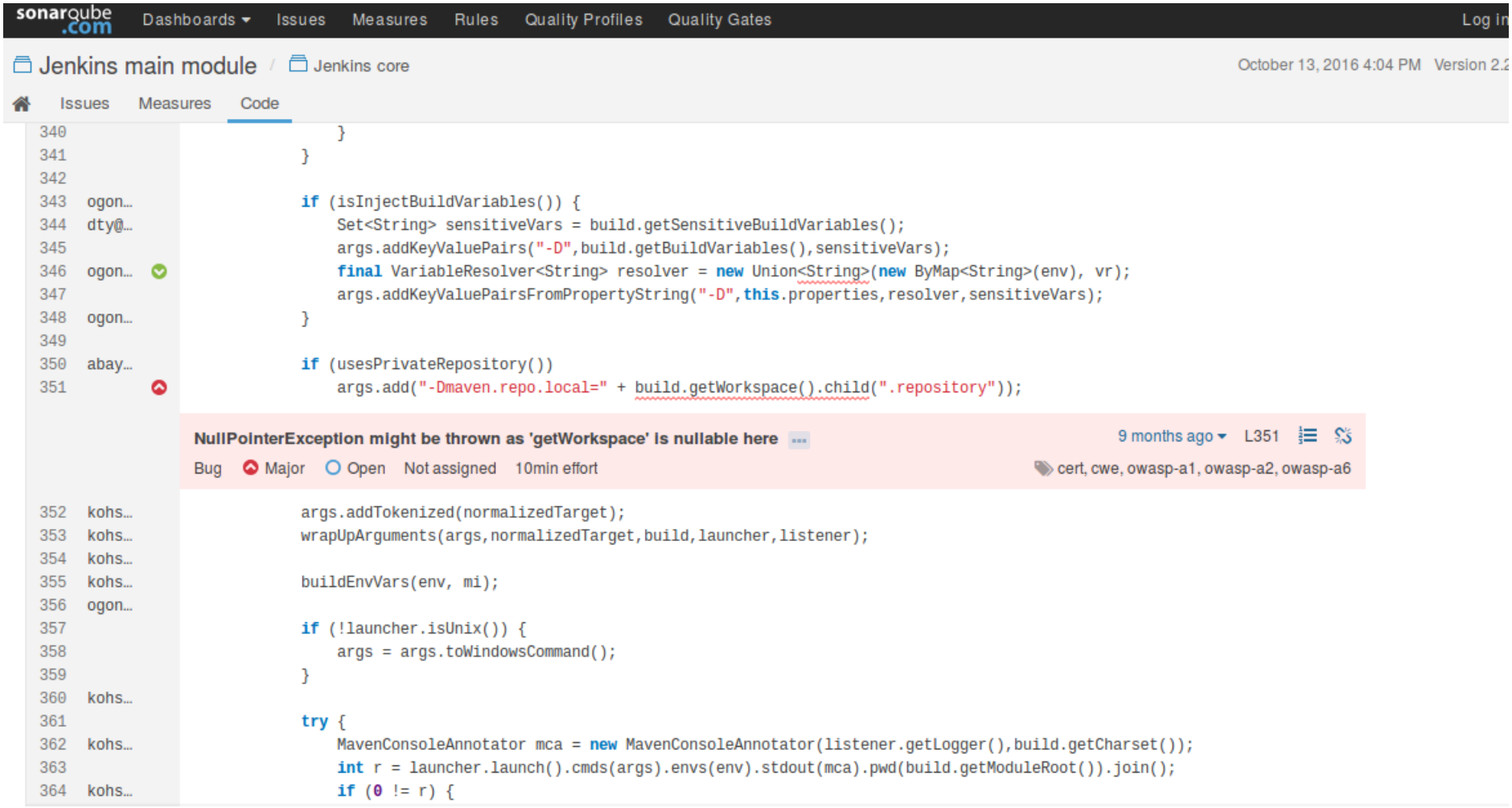
Static code analysis

- Tools:
  - e.g., Fortify Static Code Analyzer, Coverity, Clang… many

Results within IDE



Example (java code)



Top 10 User Mistakes with Static Analysis

- 10) Developers not included in process evolution
- 9) Wrong expectations
- 8) Taking an audit approach
- 7) Starting with too many rules
- 6) Workflow integration
- 5) Lack of sufficient training
- 4) No defined process
- 3) No automated process enforcement
- 2) Lack of a clear policy
- 1) Lack of management buy-in

Dynamic Code analysis

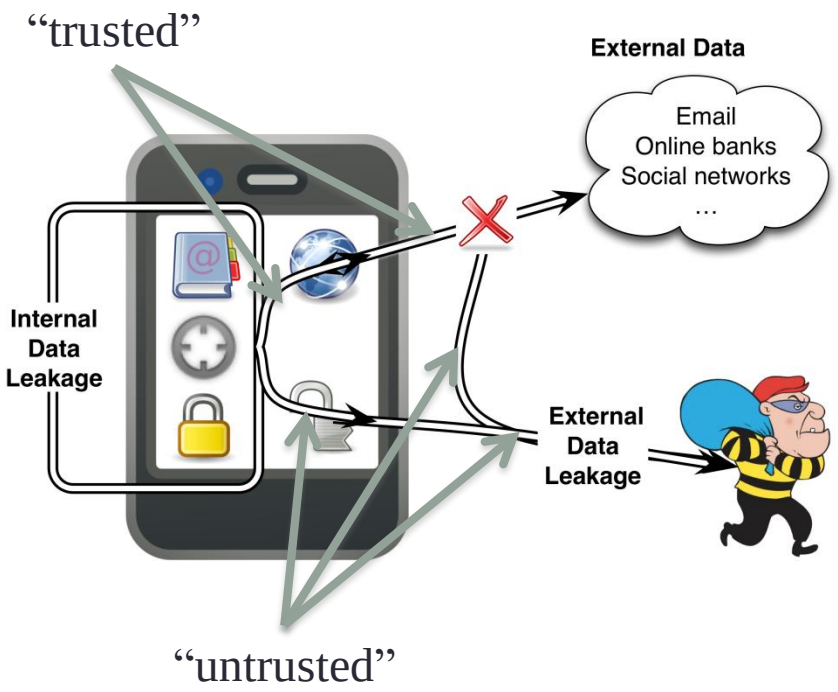
- Follow execution of a program to make security assertions
- Tools: e.g. Valgrind

(most known for finding memory leaks)

Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build

Taint analysis

- Through taint analysis one tracks each information flow among internal, external sources
  - Each flow is *tagged*, e.g., “untrusted”
  - Tag propagated as information flows among internal, external sources
  - Sound alarm if data sent to third party
- Challenges
  - Reasonable runtime, space overhead
  - Many information sources



Information leakage

Example: TaintCheck

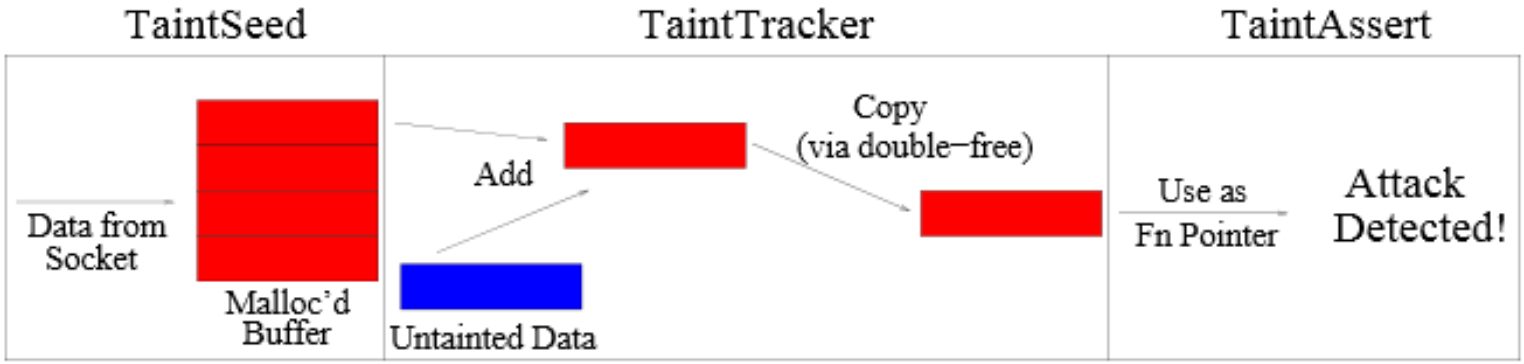
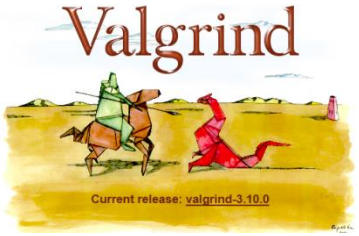


Figure 1. TaintCheck detection of an attack. (Exploit Analyzer not shown).

Valgrind use: Modifying Executables at Runtime

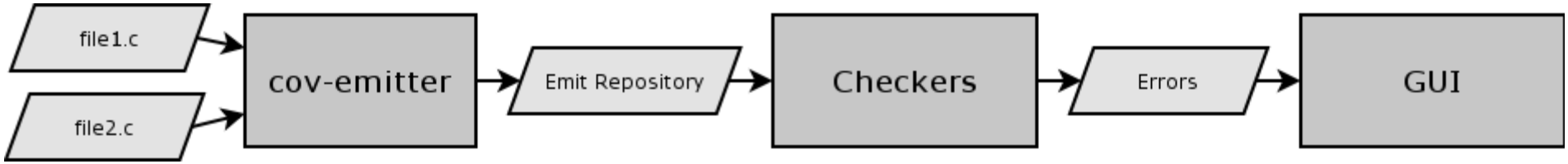
- Run executable under Valgrind system
- Give Valgrind instructions on how to instrument executable
  - literally, what instructions or function calls to search for, and what instructions to add to them
- Valgrind's processing loop:
  - Fetch next basic block of program (dictated by IP/PC)
  - Translate code into UCode, Valgrind's instruction set
  - Add instrumentation code to Valgrind UCode
  - Translate code back to x86; cache for reuse
  - Execute instrumented x86 basic block
  - Repeat…



Verification of coding practices: tools

- Parse source code and indicate (potential) bad code that can lead to exploits

- Commercial:
  - Coverity, …



- Freeware:
  - ITs4, Clang, Tripwire, MOPS

Testing

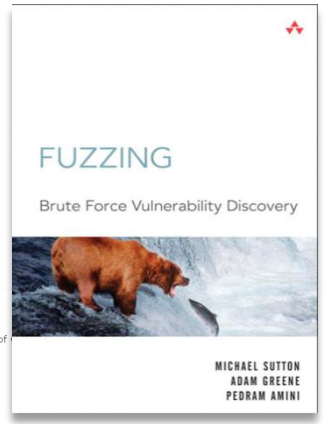
- Determine attack surface, recall attack trees
- Verification plan
- Test tools
- Perform Fuzz / Robustness Testing
- Penetration testing

Fuzzing

- There exist to day many tools that in a kind of smart way through random mutations tests implementations to see if the given tests result in faults.
  (one should be aware though that we likely find only simple faults)

- Format fuzzing (packet formats)
- Protocol fuzzing
- Fuzz frameworks
  (e.g Peach)

www.fuzzing.org
(related to a book on fuzzing)



- antiparser
  - Written in Python, simple and limited fuzzing framework.
- Autodafe
  - Can be perceived as a more powerful version of SPIKE. It's main contribution is the introduction of a UNIX-based debugging agent capable of
- AxMan
  - A web-based ActiveX fuzzing engine written by HD Moore.
- bugger
  - A Linux in-process fuzzer written by Michal Zalewski.
- COMRaider
  - A Windows GUI fuzzer written by David Zimmer, designed to fuzz COM Object Interfaces.
- Dfuz
  - Written in C, exposes a custom and easy to use scripting language for fuzzer deveopment.
- DOM-Hanoi
  - Written by H D Moore and Aviv Raff, DOM-Hanoi is designed to identify common DHTML implementation flaws by adding/removing DOM elements
- Evolutionary Fuzzing System (EFS)
  - A fuzzer which attempts to dynamically learn a protocol using code coverage and other feedback mechanisms.
- FileH
  - A haskell-based file fuzzer that generates mutated files from a list of source files and feeds them to an external program in batches.
- FileP
  - A python-based file fuzzer that generates mutated files from a list of source files and feeds them to an external program in batches.
- Fuzzled
  - A Perl based generic fuzzing framework.
- General Purpose Fuzzer (GPF)
  - Written in C, GPF has a number of modes ranging from simple pure random fuzzing to more complex protocol tokenization.
- hamachi
  - Written by H D Moore and Aviv Raff, Hamachi will look for common DHTML implementation flaws by specifying common "bad" values for method arguments and property values.
- Malybuzz
  - A Python tool focused in discovering programming faults in network software.
- mangleme
  - An automated broken HTML generator and browser tester, originally used to find dozens of security and reliability problems in all major Web browsers.
- Peach
  - Written in Python, an advanced and robust fuzzing framework which successfully separates and abstracts relevant concepts. Learning curve is a bit overwhelming.
- Protocol Informatics
  - Slides, whitepaper and code from the last publicly seen snapshot from Marshall Beddoe's work.
- QueFuzz
  - Small fuzzer that uses libnetfilter_queue to take in packets from iptables. It's fuzzing engine either randomly fuzzes binary or ASCII protocols or uses a basic fuzzing template to search and replace packet data.
- Schemer
  - XML driven generic file and protocol fuzzer.
- SMUDGE
  - Pure Python network protocol fuzzer from nd@felincemenace.
- SPIKE
  - Written in C, exposes a custom API for fuzzer development. Probably the most widely used and popular framework.
- TAOF (The Art of Fuzzing)
  - Written in Python, a cross-platform GUI driven network protocol fuzzing environment for both UNIX and Windows systems.

Note that fuzzing can be used as a tool to find exploits

SW Technology

- Use current compiler toolset
- Use Static Analysis Tools
  - Static analysis tools are now commonly used find common vulnerability types.
  - Static code analysis tools can help to ensure coding mistakes are caught and corrected as soon as possible.

Proving programs are correct

- Look at video **Rustan Leino, Microsoft Research - Program Verification: Yesterday, Today, Tomorrow**

  Floyd, flow graph

  Hoare triple, {P} S {Q}, loop invariant

Tools to proof correctness by annotating code

It is coming

Summarizing

- Writing secure software is a challenge but we have today pretty good ways that do help sw developpers!

Section	Practice
Secure Design Principles	Threat Modeling
	Use Least Privilege
	Implement Sandboxing
Secure Coding Practices	Minimize Use of Unsafe String and Buffer Functions
	Validate Input and Output to Mitigate Common Vulnerabilities
	Use Robust Integer Operations for Dynamic Memory Allocations and Array Offsets
	Use Anti-Cross Site Scripting (XSS) Libraries
	Use Canonical Data Formats
	Avoid String Concatenation for Dynamic SQL Statements
	Eliminate Weak Cryptography
	Use Logging and Tracing
Testing Recommendations	Determine Attack Surface
	Use Appropriate Testing Tools
	Perform Fuzz / Robustness Testing
	Perform Penetration Testing
Technology Recommendations	Use a Current Compiler Toolset
	Use Static Analysis Tools

- Combination of Agile sw development and approaches to secure sw development does not simply go together.
  - it requires additional effort to introduce and maintain a security perspective in Agile sw development