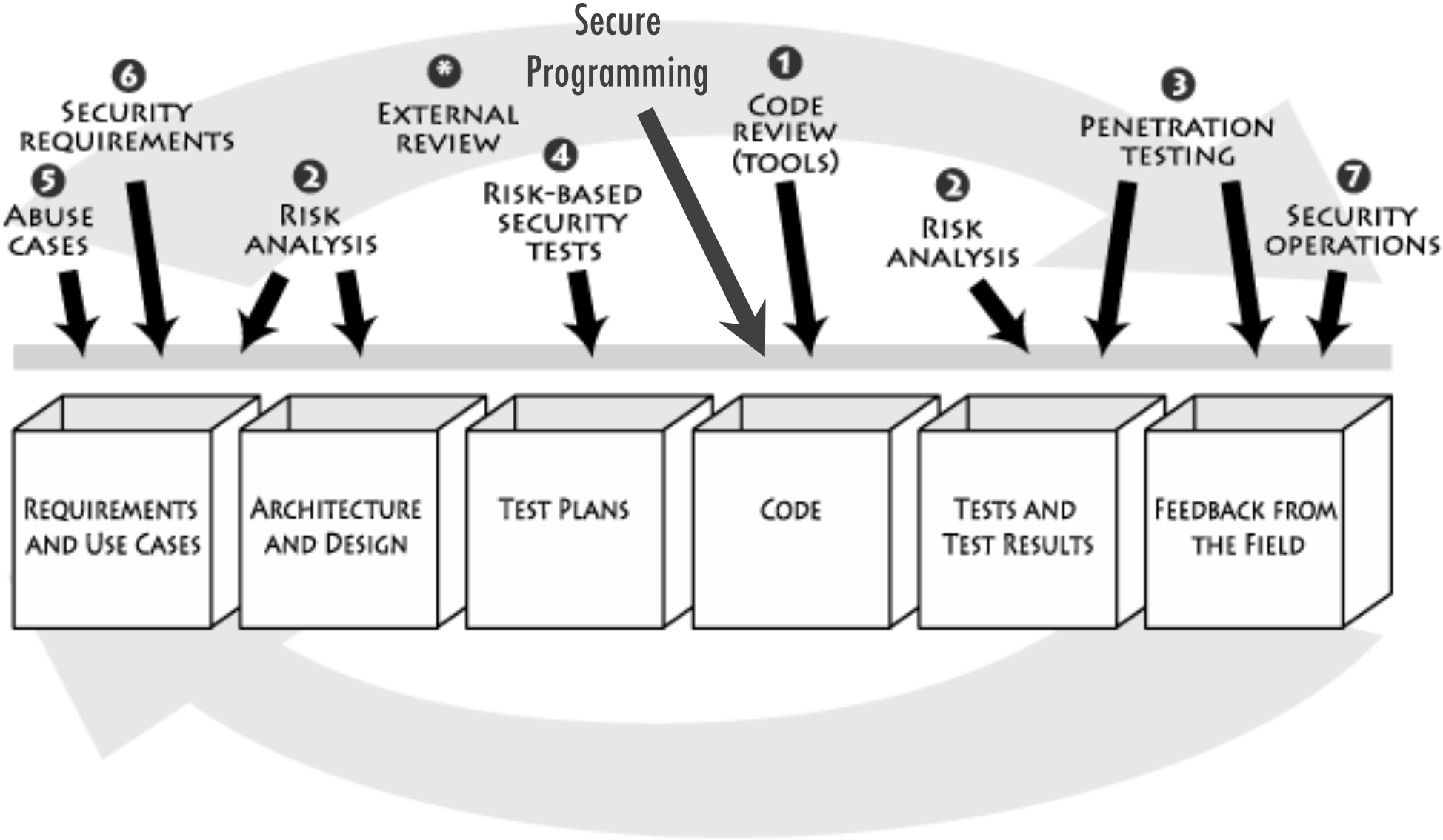
Secure Software Development Life Cycle

Secure SDLC


Secure Programming
6 Security Requirements
5 Abuse Cases
2 Risk Analysis
* External Review
4 Risk-Based Security Tests
1 Code Review (Tools)
2 Risk Analysis
3 Penetration Testing
7 Security Operations
Requirements and Use Cases
Architecture and Design
Test Plans
Code
Tests and Test Results
Feedback from the Field

Security Requirements Engineering: Two Approaches

- Security By Certification
  - Best Practices, Security Guidelines
  - Information Flow Control
  - Static & Dynamic Analysis

Secure Programming Guidelines
Code Scanners / Static Analysis / Code Audit
Security testing
Common Criteria / EAL

- Security By Design
  - Security Objectives
  - Threat Analysis

Security Architectures
Security Properties
Access Control / Cryptographic Protocols

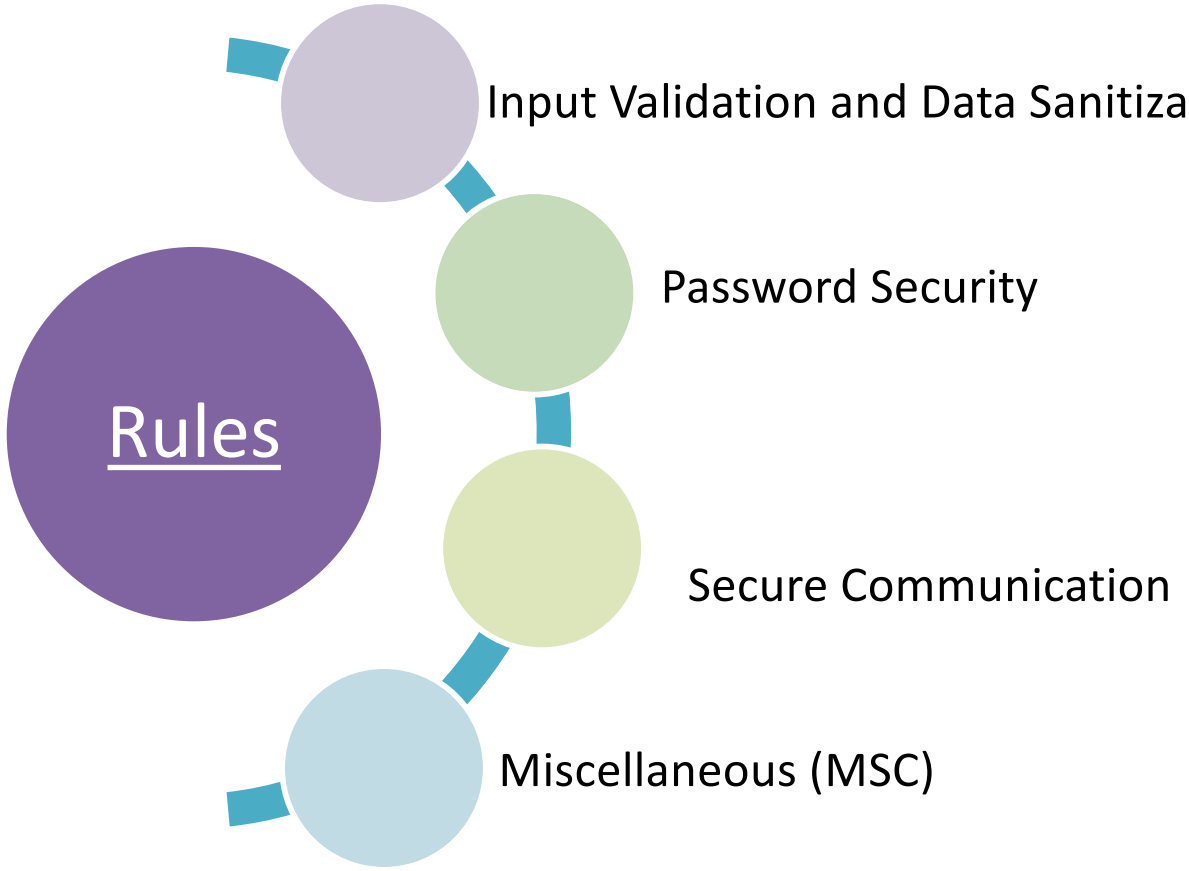
Security Requirements Engineering: Two Approaches

- Security By Certification
  - Best Practices, Security Guidelines
  - Information Flow Control
  - Static & Dynamic Analysis
- Security By Design
  - Security Objectives
  - Threat Analysis

Security Knowledge Sources

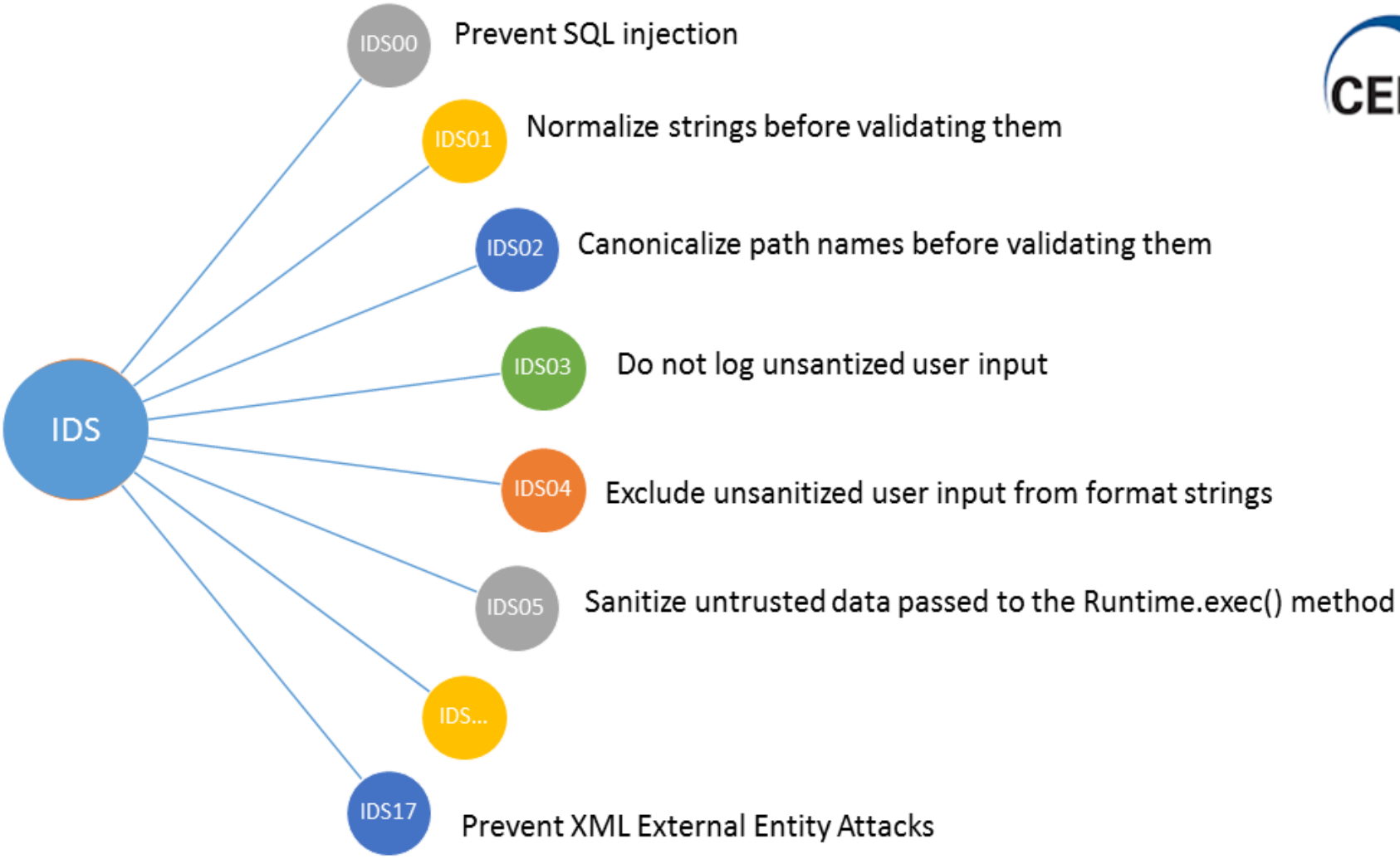
- Textbooks
- Guidelines
- Standards
- Checklists
- Documentation from past projects
- Security Design Patterns
- Structured Catalogues & Knowledge Bases

Best Practices / Secure Programming



Source: https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

Best Practices / Secure Programming



Security Guideline: Objective + Anti-Pattern

MSC62-J. Store passwords using a hash function

Créée par Matthew Wiethoff, dernière modification par David Svoboda le mai 13, 2016

Programs that store passwords as cleartext (unencrypted text data) risk exposure of those passwords in a variety of ways. Although programs generally receive passwords from users as cleartext, they should ensure that the passwords are not stored as cleartext.

An acceptable technique for limiting the exposure of passwords is the use of *hash functions*, which allow programs to indirectly compare an input password to the original password string without storing a cleartext or decryptable version of the password. This approach minimizes the exposure of the password without presenting any practical disadvantages.

Cryptographic Hash Functions

The value produced by a hash function is the *hash value* or *message digest*. Hash functions are computationally feasible functions whose inverses are computationally infeasible. In practice, a password can be encoded to a hash value, but decoding remains infeasible. The equality of passwords can be tested through the equality of their hash values.

A good practice is to always append a *salt* to the password being hashed. A salt is a unique (often sequential) or randomly generated piece of data that is stored along with the hash value. The use of a salt helps prevent brute-force attacks against the hash value, provided that the salt is long enough to generate sufficient entropy (shorter salt values cannot significantly slow down a brute-force attack). Each password should have its own salt associated with it. If a single salt were used for more than on password, two users would be able to see whether their passwords are the same.

The choice of hash function and salt length presents a trade-off between security and performance. Increasing the effort required for effective brute-force attacks by choosing a stronger hash function can also increase the time required to validate a password. Increasing the length of the salt makes brute-force attacks more difficult but requires additional storage space.

Java's `MessageDigest` class provides implementations of various cryptographic hash functions. Avoid defective functions such as the Message-Digest Algorithm (MD5). Hash functions such as Secure Hash Algorithm (SHA)-1 and SHA-2 are maintaine by the National Security Agency and are currently considered safe. In practice, many applications use SHA-256 because this hash function has reasonable performance while still being considered secure.

Noncompliant Code Example

This noncompliant code example encrypts and decrypts the password stored in `password.bin` using a symmetric key algorithm:

```
public final class Password {
  private void setPassword(byte[] pass) throws Exception {
    // Arbitrary encryption scheme
    bytes[] encrypted = encrypt(pass);
    clearArray(pass);
    // Encrypted password to password.bin
    saveBytes(encrypted,"password.bin");
    clearArray(encrypted);
  }

  boolean checkPassword(byte[] pass) throws Exception {
```

Security Guideline: Compliant Example

Compliant Solution

This compliant solution addresses the problems from the previous noncompliant code example by using a `byte` array to store the password:

```
import java.security.GeneralSecurityException;
import java.security.SecureRandom;
import java.security.spec.KeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

final class Password {
  private SecureRandom random = new SecureRandom();

  /* Set password to new value, zeroing out password */
  void setPassword(char[] pass)
      throws IOException, GeneralSecurityException  {
    byte[] salt = new byte[12];
    random.nextBytes(salt);
    saveBytes(salt, "salt.bin");
    byte[] hashVal = hashPassword( pass, salt);
    saveBytes(hashVal,"password.bin");
    Arrays.fill(hashVal, (byte) 0);
  }
  /* Indicates if given password is correct */
  boolean checkPassword(char[] pass)
      throws IOException, GeneralSecurityException  {
    byte[] salt = loadBytes("salt.bin");
    byte[] hashVal1 = hashPassword( pass, salt);
    // Load the hash value stored in password.bin
    byte[] hashVal2 = loadBytes("password.bin");
    boolean arraysEqual = timingEquals( hashVal1, hashVal2);
    Arrays.fill(hashVal1, (byte) 0);
    Arrays.fill(hashVal2, (byte) 0);
    return arraysEqual;
  }
```

Log Injection

*This is an **Attack**. To view all attacks, please see the Attack Category page.*

Last revision (mm/dd/yy): **06/6/2016**

Description

Writing unvalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs.

Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.
2. The data is written to an application or system log file.

Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.

Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act [1]. In the worst case, an attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility [2].

Examples

The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```
...
String val = request.getParameter("val");
try {
        int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
        log.info("Failed to parse val = " + val);
}
...
```

If a user submits the string "twenty-one" for val, the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

However, if an attacker submits the string "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy", the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

Description:

"Writing unvalidated user input to log files can allow an attacker to forge log entries […] Log forging vulnerabilities occur when:
1) Data enters an application from an untrusted source. […]."

Compliant Code Example:

Positive pattern

Source:
https://www.owasp.org/index.php/Log_Injection

Input Validation

This is a control. To view all control, please see the Control Category page.

This article is a stub. You can help OWASP by expanding it or discussing it on its Ta

This page contains draft content that has never been finished. Please help OWASP update this content! See FixME.
Last revision (yyyy-mm-dd): 2016-07-27
Comment: Content is poor

Using black and/or white lists which defines valid input data. Such approach is more accurate and provides better risk analysis, when there is need of modification of the lists.

E.g. When we expect digits as an input, then we should perform accurate input data validation.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char **argv)
{
        char a[256];
        strncpy(a, argv[1], sizeof(a)-1);

        int b=0;

        for(b=0; b<strlen(a); b++) {
                if(isdigit((int)a[b])) printf("%c", a[b]);
        }

        printf("\n");
        return 0;
}
```

In PHP for input data validation we may use e.g. preg_match() function:

```
<?php
  $clean = array();
  if (preg_match("/^[0-9]+:[X-Z]+$/D", $_GET['var'])) {
```

Description:

"Using black and/or white lists which defines valid input data. Such approach is more accurate and provides better risk analysis, when there is need of modification of the lists".

Compliant Code Example:

Positive pattern

Source:

https://www.owasp.org/index.php/Log_Injection

Problems with Security Guidelines

- Abstract and imprecise descriptions
- Informal specification – how to verify compliance?
- Subject to misinterpretation for developer
- Often programming language specific
- Multiple overlapping catalogs (CERT, OWASP, …)

- These are supported by code scanners which implement ad-hoc compliance checks through static analysis
  - Problem: no description of the verifications performed

Defensive Programming

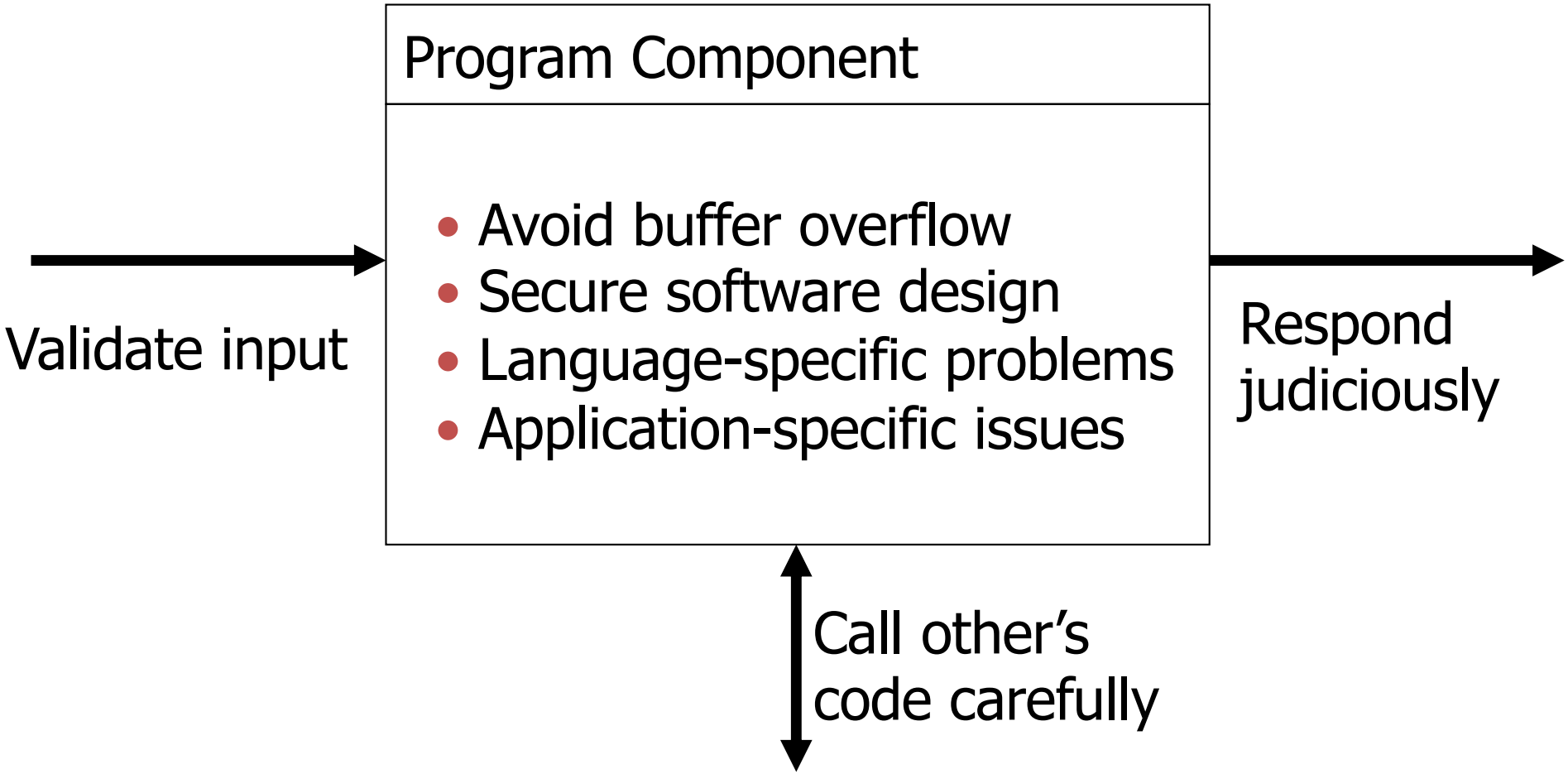
Principles [Viega and McGraw]

- Secure the weakest link
- Practice defense in depth
- Fail securely
- Follow the principle of least privilege
- Compartmentalize
- Keep it simple
- Promote privacy
- Remember that hiding is hard
- Be reluctant to trust
- Use your community resources

Secure the weakest link

- Think about possible attacks
  - How would someone try to attack this?
  - What would they want to accomplish?
- Find weakest link(s)
  - Crypto library is probably pretty good
  - Is there a way to work around crypto?
    - Data stored in encrypted form; where is *key* stored?
- Main point
  - Do security analysis of the whole system
  - Spend your time where it matters

General categories



Checking secure software

- Many rules for writing secure code
  - "sanitize user input before using it"
  - "check permissions before doing operation X"
- How to find errors?
  - Formal verification
    - + rigorous
    - – costly, expensive. *Very* rare for software
  - Testing:
    - + simple, few false positives
    - – requires running code: doesn't scale & can be impractical
  - Manual inspection
    - + flexible
    - – erratic & doesn't scale well.

Two options

- Dynamic analysis
  - Run code, possibly under instrumented conditions, to see if there are likely problems
- Static analysis
  - Inspect code or run automated method to find errors or gain confidence about their absence

Static vs Dynamic Analysis

- Dynamic
  - Need to choose sample test input
  - Can find bugs vulnerabilities
  - Cannot prove their absence
- Static
  - Consider all possible inputs (in summary form)
  - Find bugs and vulnerabilities
  - Can prove absence of bugs, in some cases

Normal Dynamic Analysis

- Run program in instrumented execution environment
  - Binary translator, Static instrumentation, emulator
- Look for bad stuff
  - Use of invalid memory, race conditions, null pointer deref, etc.
- Examples: Purify, Valgrind, Normal OS exception handlers (crashes)

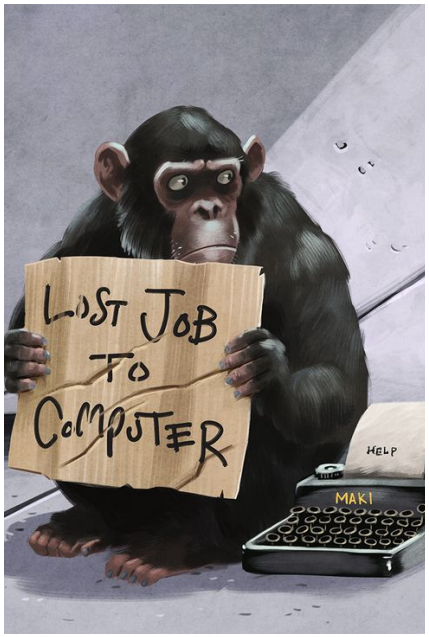
Regression vs. Fuzzing

- Regression: Run program on many normal inputs, look for badness.
  - Prevent normal users from encountering errors (e.g. assertions bad).

- Fuzzing: Run program on many abnormal inputs, look for badness.
  - Prevent attackers from encountering exploitable errors (e.g. assertions often ok)

Fuzzing Basics

- Automatically generate test cases
- Many slightly anomalous test cases are fed into a target interface
  - API or “protocol”
- Application is monitored for errors
  - Monitoring may require instrumentation
- Inputs are
  - file based (.pdf, .png, .wav, .mpg)
  - network based (http, SNMP, SOAP)
  - other (e.g. stress testing with `crashme()`)



Trivial Example

- Standard HTTP GET request
  - GET /index.html HTTP/1.1
- Anomalous requests
  - AAAAAA...AAAA /index.html HTTP/1.1
  - GET ///////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1

Different Ways To Generate Inputs

- Mutation Based - “Dumb Fuzzing”
- Generation Based - “Smart Fuzzing”

Mutation Based Fuzzing

- Little or no knowledge of the structure of the inputs
- Anomalies are added to existing valid inputs
- Anomalies may be
  - completely random
  - or follow some heuristics (e.g. remove termination, shift character forward)

Mutation Fuzzing: Pros & Cons

- Strengths
  - Simple to setup and (rather) simple to automate
  - Little to no “protocol” knowledge required
- Weaknesses
  - Limited by initial corpus
  - May fail for protocols with checksums, those which depend on challenge response, etc.

Generation Based Fuzzing

- Test cases are generated from some description of the format
  - RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of “protocol” should give better results than random fuzzing

Generation Based Fuzzing: Pros & Cons

- Strengths
  - completeness
  - Can deal with complex dependencies (e.g. checksums)

- Weaknesses
  - Need specification of protocol
    - Often can find good tools for existing protocols
  - Writing generator can be labor intensive for complex protocols
  - The specification is not the code (implementation may correspond to incomplete or wrongly interpreted specification)

Injecting Input

- Simplest
  - Run program on fuzzed file (file fuzzing)
  - Replay fuzzed packet trace (network fuzzing)
- Modify existing program/client
  - Code invokes the fuzzer at the appropriate point
- Use fuzzing framework
  - e.g. Peach automates the generation of COM interface fuzzers

Problem Detection

- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (valgrind/purify)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Roll your own checker e.g. valgrind skins

How Much Fuzz Is Enough?

- Mutation based fuzzers can generate an infinite number of test cases... When has the fuzzer run long enough?
- Generation based fuzzers generate a finite number of test cases. What happens when they're all run and no bugs are found?

Code Coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric which can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. `gcov`

Types of Code Coverage

- Line coverage
  - Measures how many lines of source code have been executed.
- Branch coverage
  - Measures how many branches in code have been taken (conditional jmps)
- Path coverage
  - Measures how many paths have been taken

Static Analysis Basics

- Model program properties abstractly, look for problems
- Tools come from program analysis
  - Type inference, data flow analysis, theorem proving
- Usually on source code, can be on byte code or disassembly
- Strengths
  - Complete code coverage (in theory)
  - Potentially verify absence/report all instances of whole class of bugs
  - Catches different bugs than dynamic analysis
- Weaknesses
  - High false positive rates
  - Many properties cannot be easily modeled
  - Difficult to build
  - Almost never have all source code in real systems (operating system, shared libraries, dynamic loading, etc.)

Two Types of Static Analysis

- (Rather) Simple code analysis.
  - Look for known code issues: e.g., unsafe string functions `strncpy(), sprintf(), gets()`
  - Look for unsafe functions in your source base
  - Look for recurring problem code (problematic interfaces, copy/paste of bad code, etc.)
- Deeper analysis
  - Requires complex code parsing and computations
  - Some are implemented in tools like coverity, fortify, visual studio …
  - Otherwise must be developed on top of parser like LLVM

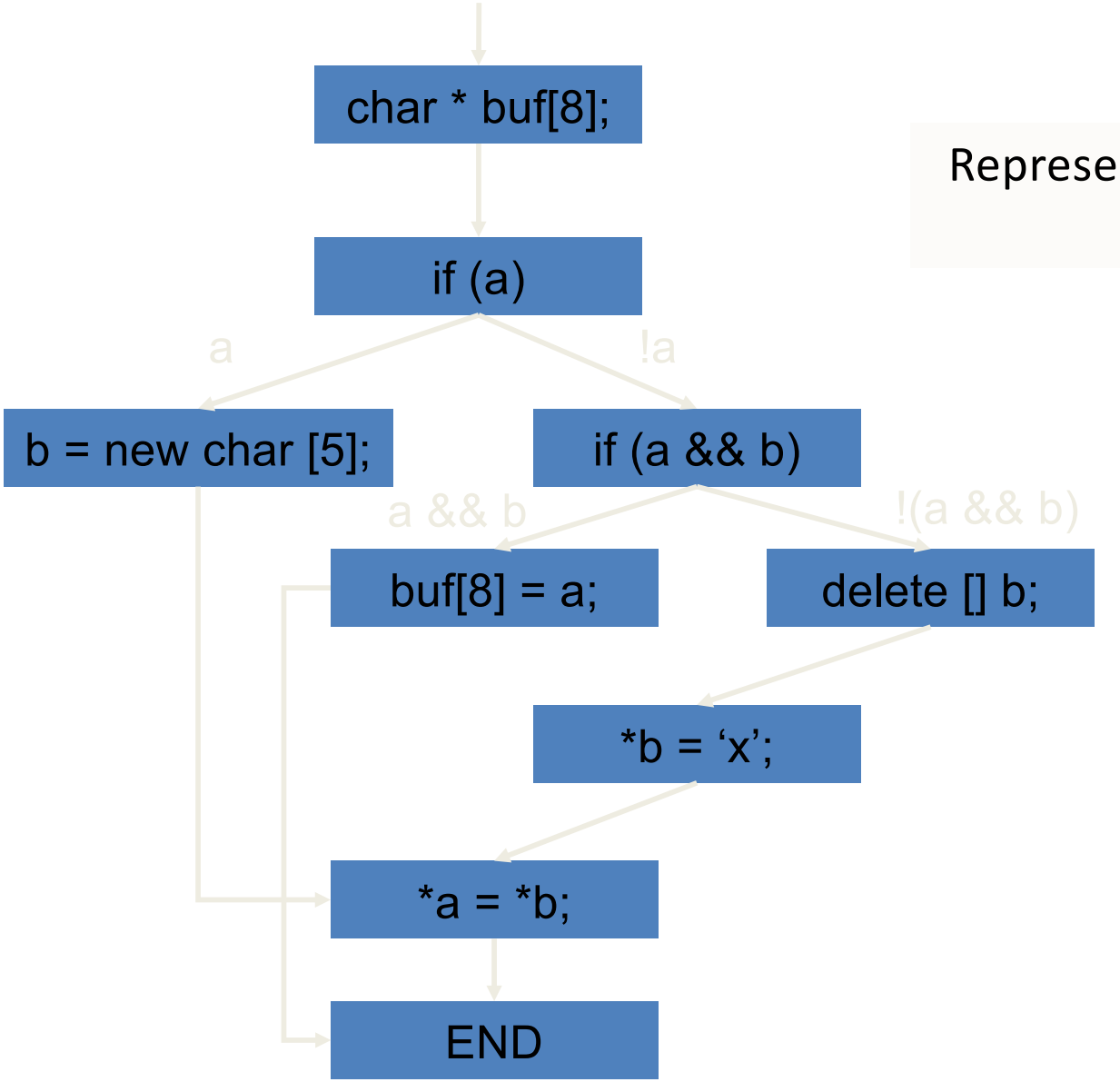
Static analysis: Soundness, Completeness

Property	Definition
Soundness	"Sound for reporting correctness" Analysis says no bugs → No bugs or equivalently There is a bug → Analysis finds a bug
Completeness	"Complete for reporting correctness" No bugs → Analysis says no bugs

Recall: $A \rightarrow B$ is equivalent to $(\neg B) \rightarrow (\neg A)$

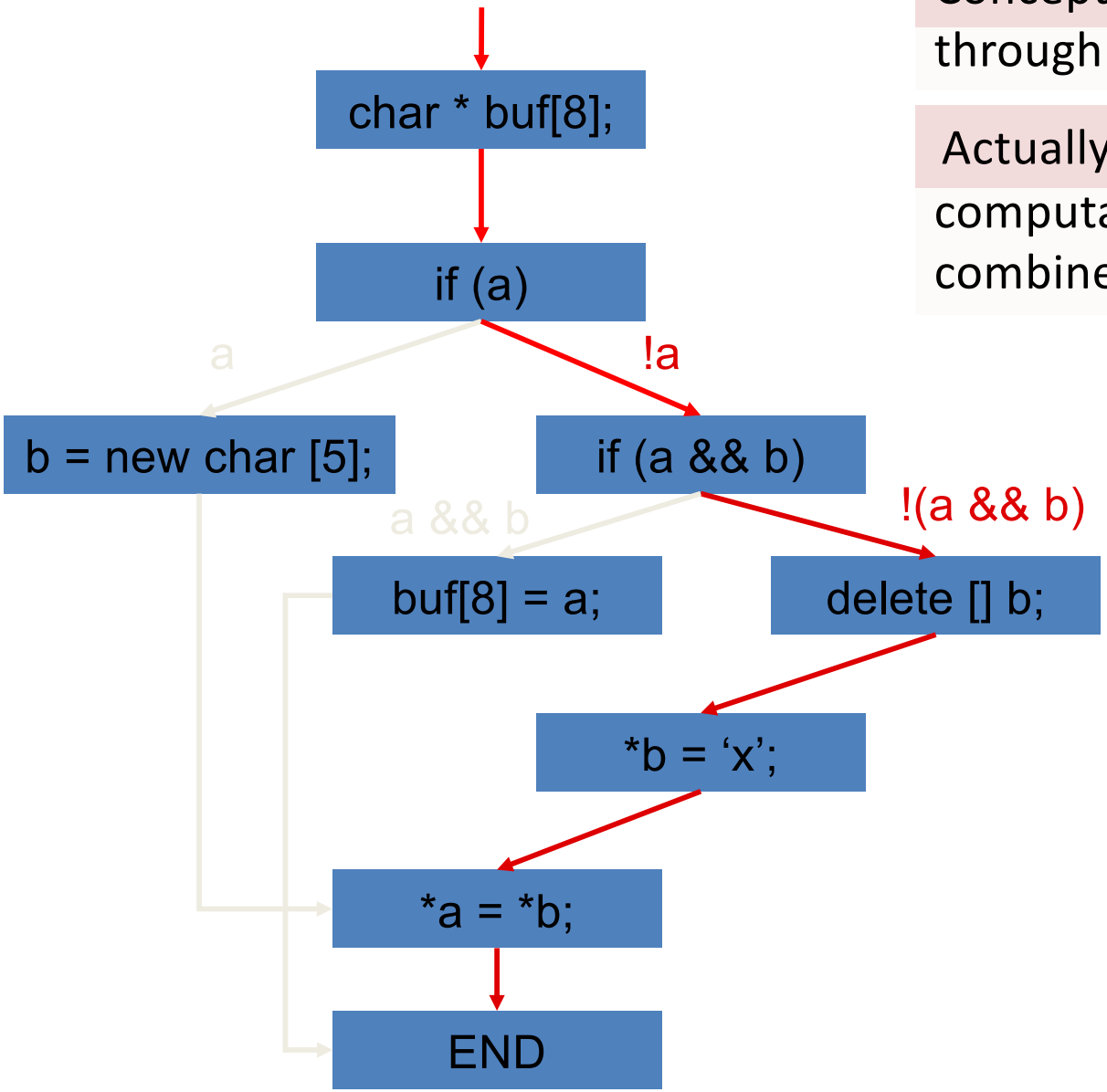
	Complete	Incomplete
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

Control Flow Graph



Represent logical structure of code in graph form

Path Traversal

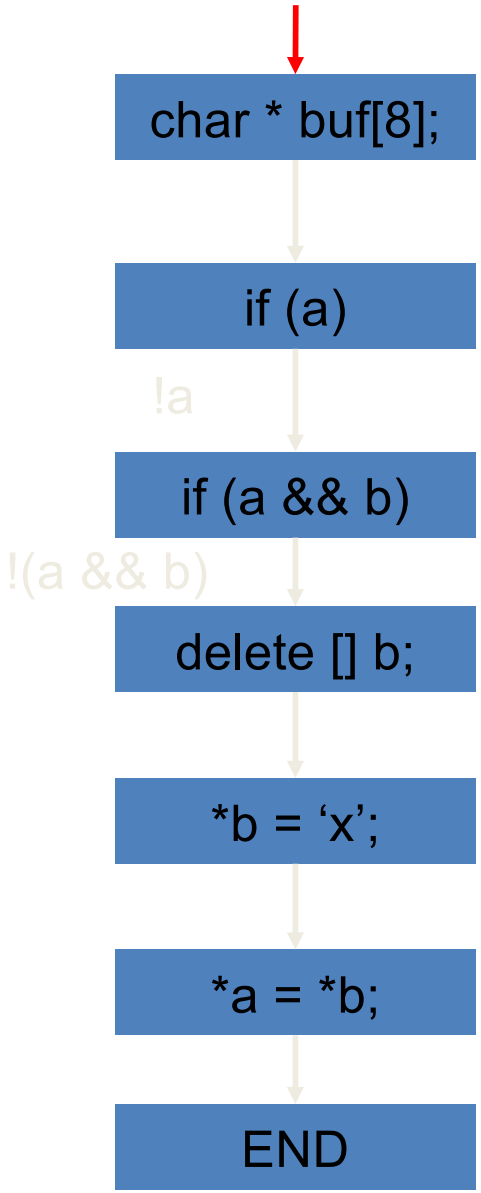


Conceptually Analyze each path through control graph separately

Actually Perform some checking computation once per node; combine paths at merge nodes

Apply Checking

Null pointers Use after free Array overrun



See how three checkers are run for this path

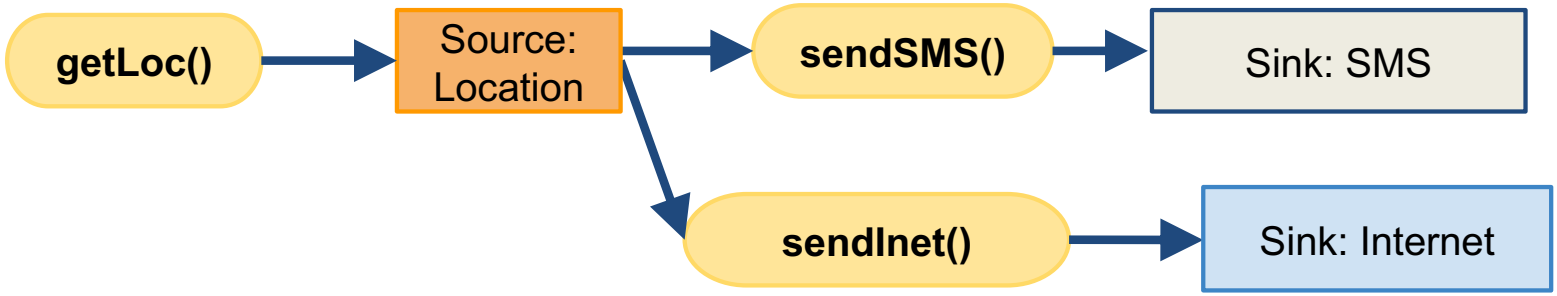
Checker

- Defined by a state diagram, with state transitions and error states

Run Checker

- Assign initial state to each program var
- State at program point depends on state at previous point, program actions
- Emit error if error state reached

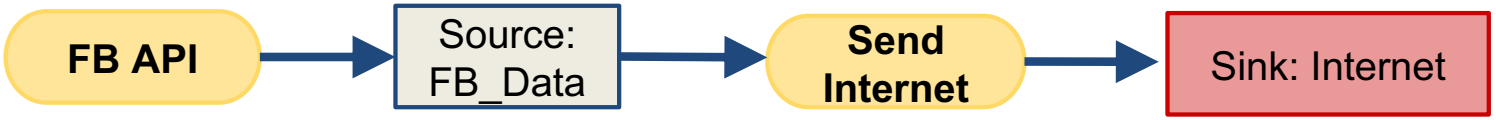
Data Flow Analysis



- Source-to-sink flows
  - Sources: Location, Calendar, Contacts, Device ID etc.
  - Sinks: Internet, SMS, Disk, etc.

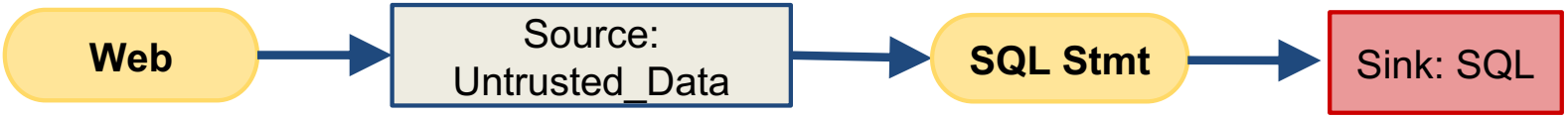
Applications of Data Flow Analysis

- Vulnerability Discovery
- Malware/Greyware Analysis
  - Data flow summaries enable enterprise-specific policies
- API Misuse and Data Theft Detection



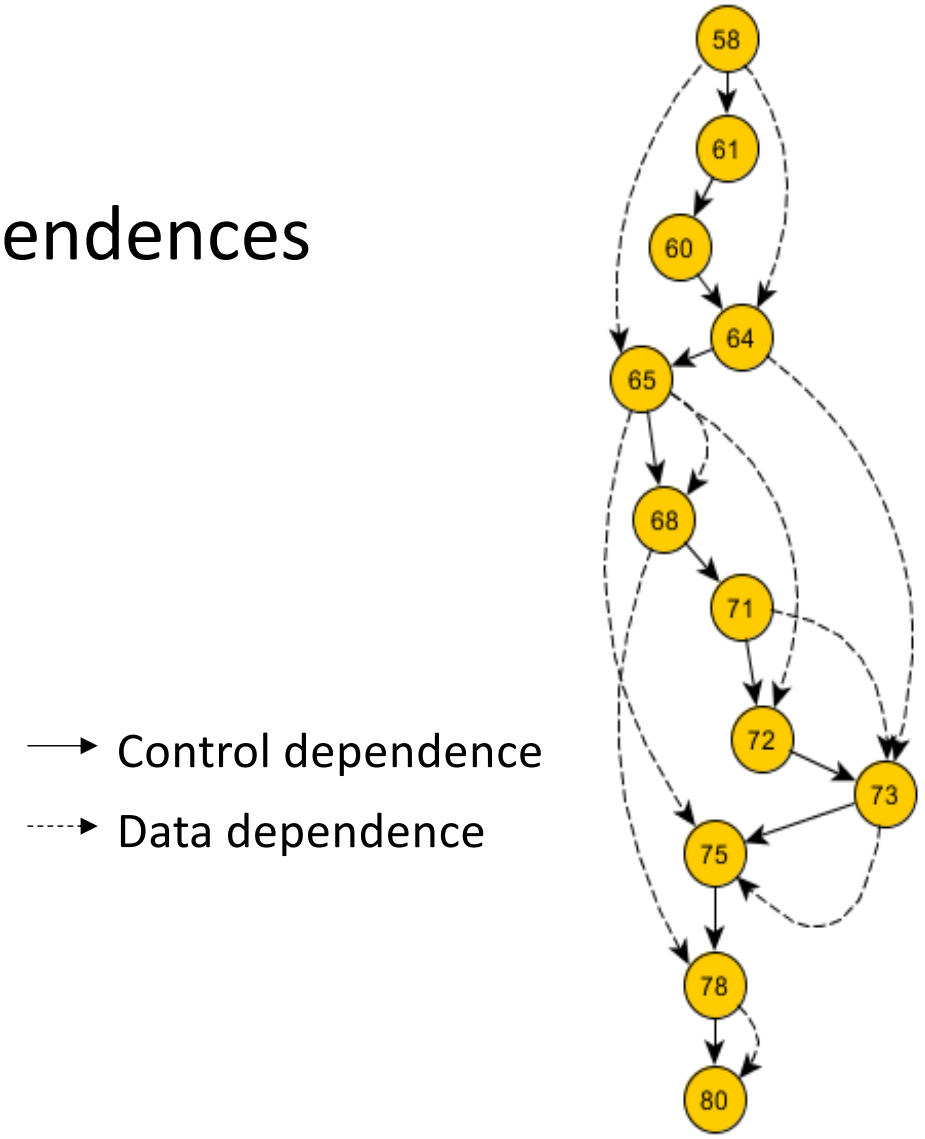
- Automatic Generation of App Privacy Policies
  - Avoid liability, protect consumer privacy

Privacy Policy
This app collects your:
Contacts
Phone Number
Address



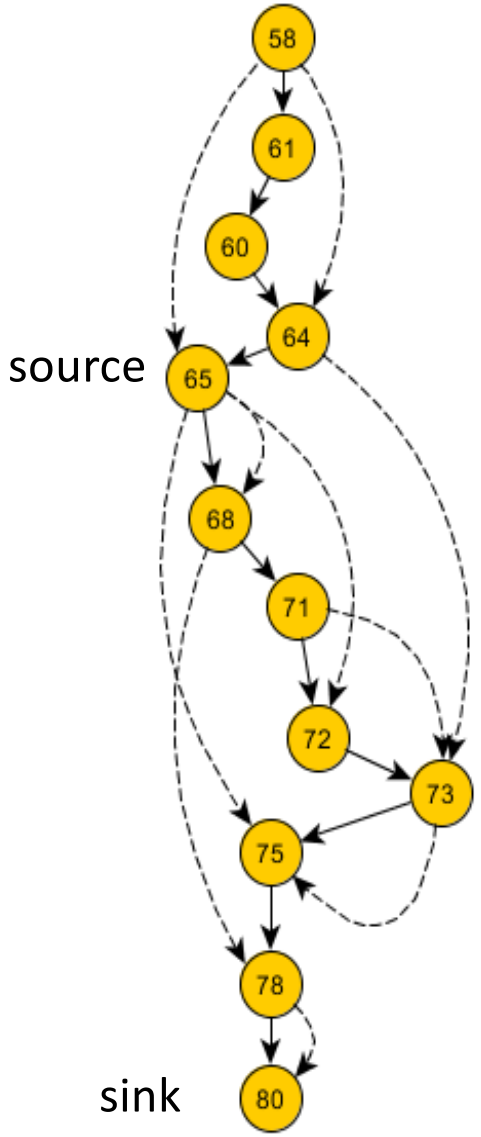
Program Dependence Graph (PDG)

- Control Dependences
- Explicit + Implicit Data Dependences
- Properties:
  - Path-sensitive
  - Context-Sensitive
  - Object-Sensitive



JOANA IFC tool

- Intended for Information Flow Analysis
- Annotations: SINK / SOURCE
- Non-Interference: Security Levels (HIGH / LOW)



Benefits: automated code analysis

- Examples
  - C language: FindBugs, Fortify, Coverity, MS tools (commercial), KLEE (academic)
  - Java language: JOANA, FindBugs, Soot, PMD (academic)

- Objectives
  - Capture recommended practices, known to experts, in tool available to all
  - Also Capture information flows, non-interference

Dynamic analysis

- Instrument code for testing
  - Heap memory: Purify
  - Perl tainting (information flow)
  - Java race condition checking
- Black-box testing
  - Fuzzing and penetration testing
  - Black-box web application security analysis

Purify

- Goal
  - Instrument a program to detect run-time memory errors (out-of-bounds, use-before-init) and memory leaks
- Technique
  - Works on relocatable object code
    - Link to modified malloc that provides tracking tables
  - Memory access errors: insert instruction sequence before each load and store instruction
  - Memory leaks: GC algorithm

Perl tainting

- Run-time checking of Perl code
  - Perl used for CGI scripts, security sensitive
  - Taint checking stops some potentially unsafe calls
- Tainted strings
  - User input, Values derived from user input
  - Except result of matching against untainted string
- Prohibited calls
  - print $form_data{"email"} . "\n";
    - OK since print is safe (???)
  - system("mail " . $form_data{"email"});
    - Flagged system call with user input as argument