Common Exploits and How to Prevent Them

David Svoboda

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Copyright 2016 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0003973

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Introduction

Security Flaws

A software defect is the encoding of a human error into the software, including omissions

A security flaw is a software defect that poses a potential security risk

Eliminating software defects eliminates security flaws

Vulnerabilities

A security policy is the definition of the security requirements for the system, for example, a statement of which resources may be accessed and how

A vulnerability is a set of conditions that allows an attacker to violate an explicit or implicit security policy

A security flaw can cause a program to be vulnerable to attack
But not all security flaws lead to vulnerabilities

Exploits

An exploit is a program or technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy

Vulnerabilities in software are subject to exploitation

Exploits can take many form including

- worms
- viruses
- trojans

Proof-of-Concept Exploits

May be developed to prove the existence of a vulnerability

Are beneficial when properly managed

In the wrong hands can be quickly transformed into a worm or virus or used in an attack

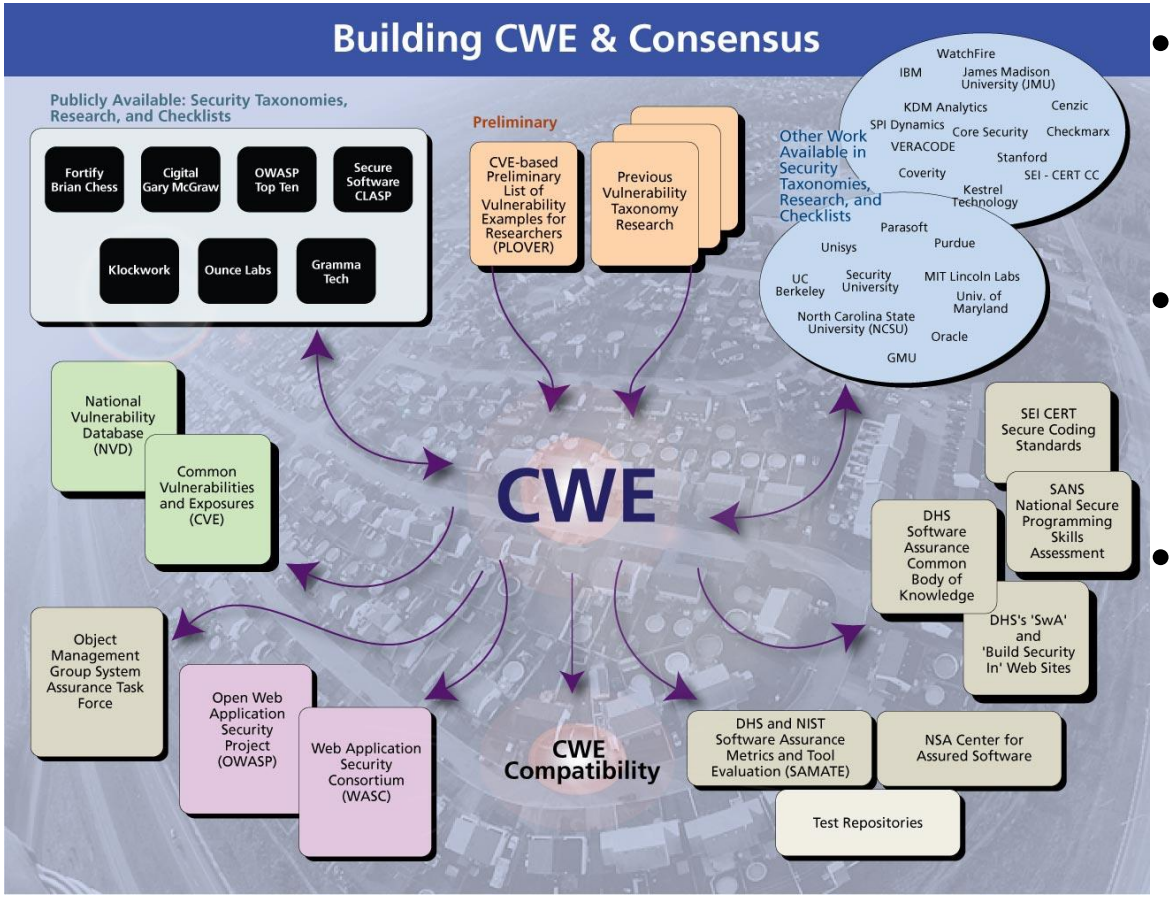
Mitigations

Mitigations are methods, techniques, processes, tools, or runtime libraries that can prevent or limit exploits against vulnerabilities

At the source code level, a mitigation may involve replacing an unbounded string copy operation with a bounded one

At a system or network level, a mitigation may involve turning off a port or filtering traffic to prevent an attacker from accessing a vulnerability

MITRE Common Weakness Enumeration (CWEs)



- Hierarchical, a "tree" of weaknesses
- Lists languages each weakness can occur in
- Simple examples of vulnerable code

https://cwe.mitre.org/

2011 CWE/SANS Top 25 Most Dangerous Software Errors

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security D

http://cwe.mitre.org/top25/#Listing

Common Vulnerabilities and Exposures

Maintained by MITRE

List of known vulnerabilities in popular software.

Cross-referenced with CWE

https://cve.mitre.org/index.html

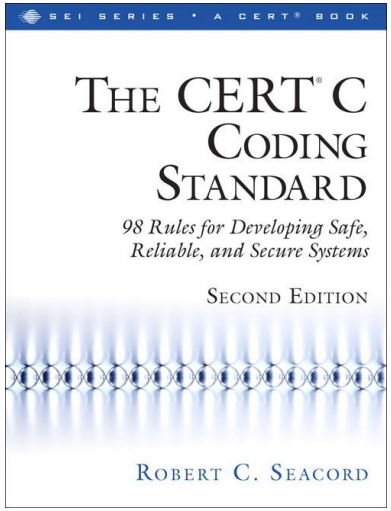
OWASP Top 10 2013

A1-Injection
A2-Broken Authentication and Session Management
A3-Cross-Site Scripting (XSS)
A4-Insecure Direct Object References
A5-Security Misconfiguration
A6-Sensitive Data Exposure
A7-Missing Function Level Access Control
A8-Cross-Site Request Forgery (CSRF)
A9-Using Components with Known Vulnerabilities
A10-Unvalidated Redirects and Forwards

https://www.owasp.org/index.php/Top_10_2013-Top_10

- Focused on web vulnerabilities

CERT Secure Coding Standards

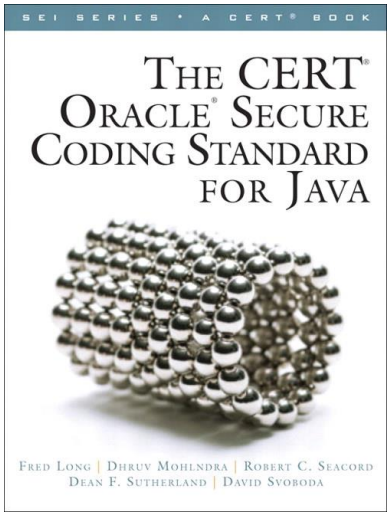


CERT C Secure Coding Standard

- Version 1.0 (C99) published in 2009
- Version 2.0 (C11) published in 2014
- ISO/IEC TS 17961 C Secure Coding Rules Technical Specification
- Conformance Test Suite

CERT C++ Secure Coding Standard

- Version 1.0 under development



CERT Oracle Secure Coding Standard for Java

- Version 1.0 (Java 7) published in 2011
- Java Secure Coding Guidelines
- Subset applicable to Android development
- Android Annex

The CERT Perl Secure Coding Standard

- Version 1.0 under development

https://www.securecoding.cert.org/confluence/x/BgE

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Injection

Injection

Malicious user input is sent to some kind of processor

AKA ‘eval’ problems

Processor	Injection Type
HTML Parser (inc. Web Browser)	Cross-Site Scripting (XSS)
Shell	OS Command
C `printf()` function family	Format String
Database	SQL
Regular Expression Library	Regex
File access function (eg `fopen()`)	Pathname
XML	XML

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

SQL Injection Demo

SQL Injection

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

A1-Injection

IDS00-J. Prevent SQL injection

Trust Boundaries

Software often contains multiple components & libraries

Each component may operate in one or more *trusted domains*

- Details of trusted domains driven by architecture, security policy, required resources, functionality, *etc.*

Example:

- Component A can access file-system, but lacks any network access
- Component B has general network access, but lacks access to the file-system and the secure network
- Component C can access a secure network, but lacks access to the file-system and the general network

Trust Boundary Security

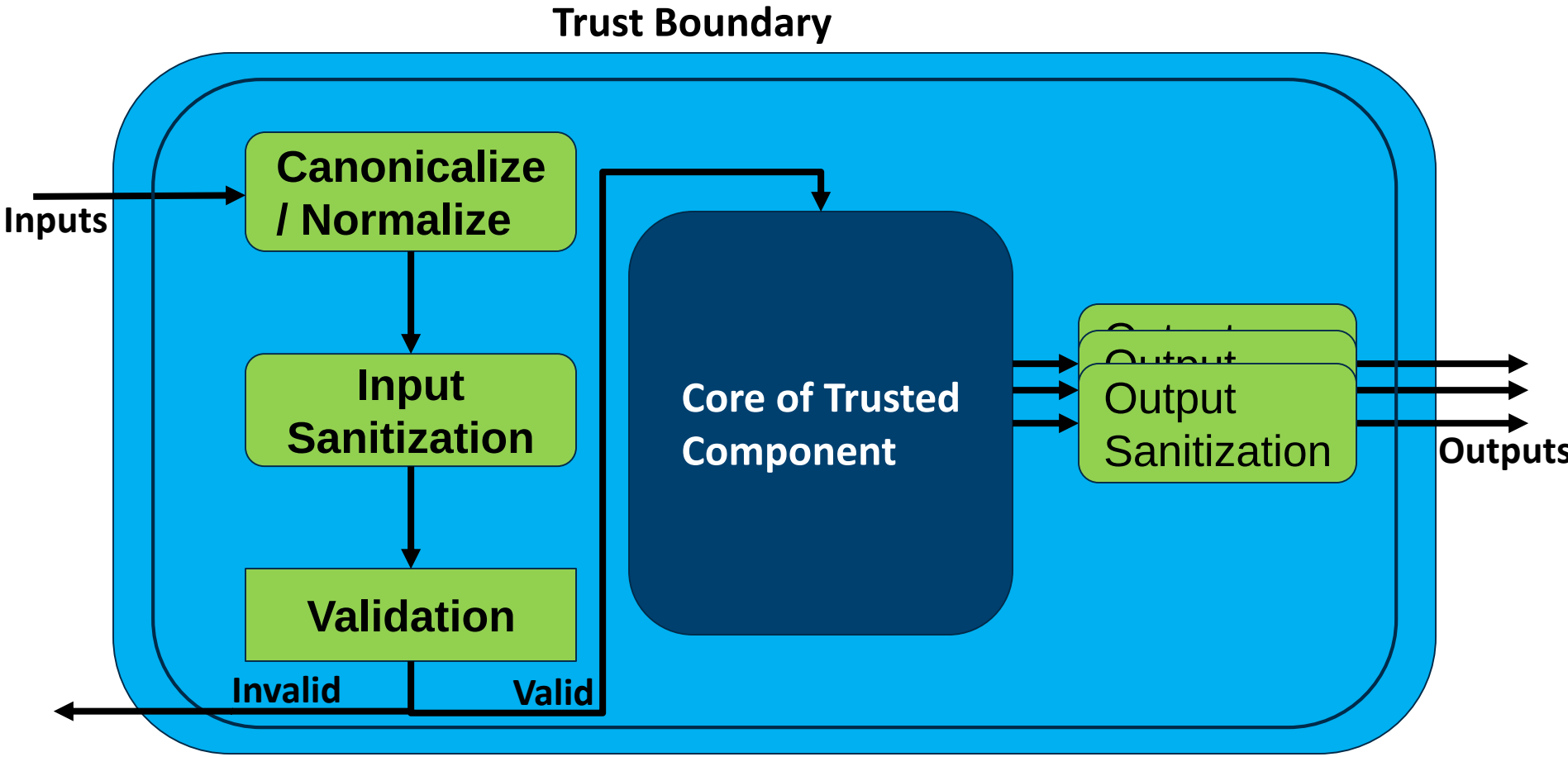
Programs must take steps to ensure that any data that crosses a trust boundary is both

- Appropriate
- Non-malicious

This can include appropriate

- Canonicalization / Normalization
- Input Sanitization
- Validation

Trust Boundary Diagram



Validation

Validation: The process of checking inputs to ensure that they fall within the intended input domain of the receiver

- Prevent errors, by disallowing invalid inputs
- Typically does not modify input

Details are specific to particular systems, inputs, *etc.*

Examples:

- Does input number fall within required numeric range?
- Can not open a nonexistent file for reading.
- Temporal properties: Unix **`sudo`** command requires authentication
  - Unless user previously authenticated within the past 30 seconds

Sanitization

Sanitization: The process of ensuring that data to be passed to a subsystem does not violate a system's security policy.

- Often converts valid-but-insecure input into invalid input.
- Applies equally to input and output.
  - Output sanitization usually prevents sensitive information leak.
- When platform provides sanitization routines, use them!

Examples:

- Elimination of unwanted characters from input string by means of removing, replacing, encoding, or escaping the characters
- Prevent user from specifying pathname to file they lack privilege to access
- Prevent user from executing Javascript or SQL

Canonicalization / Normalization

Useful precursor to string validation.

Canonicalization: Reducing the input to its simplest equivalent known form (aka canonical form)

Examples:

- Resolving `./` or `../` in path names and URLs
- Conversion of case-insensitive strings to lowercase

Normalization: Conversion of input to a standard form (not necessarily simplest)

Examples:

- Unicode conversion (to NFKC or NFKD)

Canonicalize / Normalize before Sanitizing / Validating 1

An application forbids `<script>` tags in its input

- Part of strategy to avoid XSS attacks

Input string could be user controlled

Recall:

- Java uses Unicode for its Characters
  - Unicode V4 in Java SE6
  - Unicode V6 in Java SE7

Canonicalize / Normalize before Sanitizing / Validating 2

Suppose the input string were

```
String s = "\uFE64" + "script" + "\uFE65";
```

- FYI: these are the Unicode 'SMALL LESS-THAN SIGN' and 'SMALL GREATER-THAN SIGN' characters
- They *aren't* the standard angle brackets
- But they *normalize* to the standard angle brackets

If normalization performed after sanitization:

- The sanitization check fails to spot them
- Then the normalization changes them to '<' and '>'

A **<script>** tag can sneak through the checking!

Vulnerable Code 1

```
boolean isPasswordCorrect(String name, char[] password)
      throws SQLException, ClassNotFoundException {
   Connection connection = getConnection();
...
   String pwd = new String(password);

   String sqlString = "SELECT * FROM Users WHERE name = '"
       + name + "' AND password = '" + pwd + "'";
   Statement stmt = connection.createStatement();
   ResultSet rs = stmt.executeQuery(sqlString);

   if (!rs.next()) {
     return false;
   }
```

Vulnerable Code 2

```
boolean isE                                              ord)
      throws
    Connecti
...
    String pwd = new String(password);

    String sqlString = "SELECT * FROM Users WHERE name = '"
        + name + "' AND password = '" + pwd + "'";
    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery(sqlS      ing);

    if (!rs.next()) {
      return false;
    }
```

password should not be stored in the database or a **java.lang.String** unencrypted. Ideally we'd hash it here so the database sees only the hashed password.

Name and **password** are never sanitized before being added to SQL command!

Secure Coding Tutorial

SQL Injection Mitigation Demo

Mitigation

```
boolean isPasswordCorrect(String name, char[] password)
      throws SQLException, ClassNotFoundException {
   Connection connection = getConnection();
...
   String pwd = new String(password);
   String sqlString =
     "SELECT * FROM Users WHERE name=? AND password=?";
   PreparedStatement stmt =
     connection.prepareStatement(sqlString);
   stmt.setString(1, name);
   stmt.setString(2, pwd);
   ResultSet rs = stmt.executeQuery();

   if (!rs.next()) {
     return false;
   }
```

Don't forget to hash `password`!

Sanitizes input by rules of this SQL parser.

SQL Injection Summary

Sanitize the input to your database queries!

- Be wary of string concatenation
  - Strings being joined may originate from different trust domains.
- Use sanitization provided by your platform (**`java.sql.PreparedStatement`**)

Don’t worry if your sanitization invalidates the input.

- Butchered input less egregious than injection!

Language-independent (specific to SQL, not Java)

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

OS Command Injection Demo

OS Command Injection

Rank	Score	ID	Name
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

A1-Injection

IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method

ENV33-C. Do not call system()

IDS31-PL. Do not use the two-argument form of open()

Vulnerable Code 1

```
perl -e 'while (<>) {print "contents: $_";}' *
```

which is equivalent to:

```
perl -n -e 'print "contents: $_";' *
```

which is equivalent to:

```
perl -p -e '$_ = "contents: $_";' *
```

Vulnerable Code 2

```
while (<>) {
  print "contents: $_";
}
```

is also shorthand for:

```
foreach (@ARGV) {
  open(my $file, $_);
  while (<$file>) {
    print "contents: $_";
  }
}
```

Vulnerable Code 3

```
while (<>) {
  print "contents: $_";
}
```

is also shorthand for:

```
foreach (@ARGV) {
  open(my $file, $_);
  while (<$file>) {
    print "contents: $_";
  }
}
```

Executes shell command if argument begins or ends with |

Perl open()

open FILEHANDLE,EXPR
open FILEHANDLE,MODE,EXPR
…
Opens the file whose filename is given by EXPR, and associates it with FILEHANDLE.
…
For three or more arguments if MODE is |- , the filename is interpreted as a command to which output is to be piped, and if MODE is -| , the filename is interpreted as a command that pipes output to us. In the two-argument (and one-argument) form, one should replace dash (-) with the command.

open (Perl 5 version 24.0 documentation)

Vulnerable Code 4

```
perl -e 'while (<>) {print "contents: $_";}' *
```

which is equivalent to:

```
perl -n -e 'print "contents: $_";' *
```

which is equivalent to:

All these forms are vulnerable too!

```
perl -p -e '$_ = "contents: $_";' *
```

Mitigation

```
use Carp;

foreach $arg (@ARGV) {
  open(my $file, "<", $arg) \
    or croak "cannot open $arg";
  while (<$file>) {
    print "contents: $_";
  }
}
```

IDS31-PL. Do not use the two-argument form of **open()**

EXP30-PL. Do not use deprecated or obsolete functions or modules (eg **die()**)

EXP32-PL. Do not ignore function return values

Sorry, no short way to write this code!

OS Command Injection Summary

Sometimes the easier code is the less secure code.

Beware obscure features!

- They may be more useful to attackers than you.

Layers of abstraction help to obscure features.

- Perl's file-open mechanism buried under
  - **<>** operator.
  - **-n** option
  - **-p** option

Use the three-argument version of **open()**

- 2nd argument indicates whether to open file or run command

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Format String Injection Demo

Format String Injection

CVE 2015-8617

Rank	Score	ID	Name
[23]	61.0	CWE-134	Uncontrolled Format String

A1-Injection

FIO30-C. Exclude user input from format strings

IDS06-J. Exclude unsanitized user input from **format** strings

IDS30-PL. Exclude user input from format strings

Format Strings 1

Format strings are character sequences consisting of ordinary characters (excluding %) and conversion specifications.

Ordinary characters are copied unchanged to the output stream.

Conversion specifications

- convert arguments according to a corresponding conversion specifier
- write the results to the output stream

Conversion specifications begin with a percent sign (%) and are interpreted from left to right.

Format Strings 2

If there are more arguments than conversion specifications, the extra arguments are ignored.

If there are not enough arguments for all the conversion specifications, the results are undefined.

Degrees of Severity

CIA Triad:

- Confidentiality
- Integrity
- Availability

CERT Severity Levels:

Severity—How serious are the consequences of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	low	denial-of-service attack, abnormal termination
2	medium	unintentional information disclosure
3	high	run arbitrary code, privilege escalation

Crashing a Program

An invalid pointer access or unmapped address read can be triggered by calling a formatted output function:

```
printf("%s%s%s%s%s%s%s%s%s%s%s");
```

The `%s` conversion specifier retrieves memory at an address specified in the corresponding argument on the execution stack.

Because no string parameters are supplied, `printf()` reads arbitrary memory locations from the stack until

- the format string is exhausted
- an invalid pointer or unmapped address is encountered

Viewing Memory Content

The `%s` conversion specifier displays memory at the address specified by the argument pointer as an ASCII string until a null byte is encountered.

If an attacker can manipulate the argument pointer to reference a particular address, the `%s` conversion specifier will output memory at that location.

The `%n` Conversion Specifier

Formatted output functions are dangerous because most programmers are unaware of their capabilities.

On platforms where `int` and addresses are the same size (such as x86-32), the ability to write an integer to an arbitrary address can be used to execute arbitrary code on a compromised system.

The `%n` conversion specifier

- was created to help align formatted output strings
- writes the number of characters successfully output to an integer address provided as an argument

Format String Injection Summary

Even popular software like PHP is insecure!

Beware obscure features!

- They may be more useful to attackers than you.

Sanitize your format strings!

- Use string literals
- Or use less powerful functions
  - `fputs()` instead of `fprintf()`

Occurs in any language with format strings

But worse in C-like languages

- because they are not memory-safe!

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Memory Corruption

Memory Corruption

All software uses memory.

Java and many newer languages protect memory from careless reads & writes:

- Array reads & writes are bounds-checked
- Memory not freed until no longer needed
- Dereferencing null pointers causes termination

C opted to be fast & efficient rather than safe.

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Buffer Overflow Demo

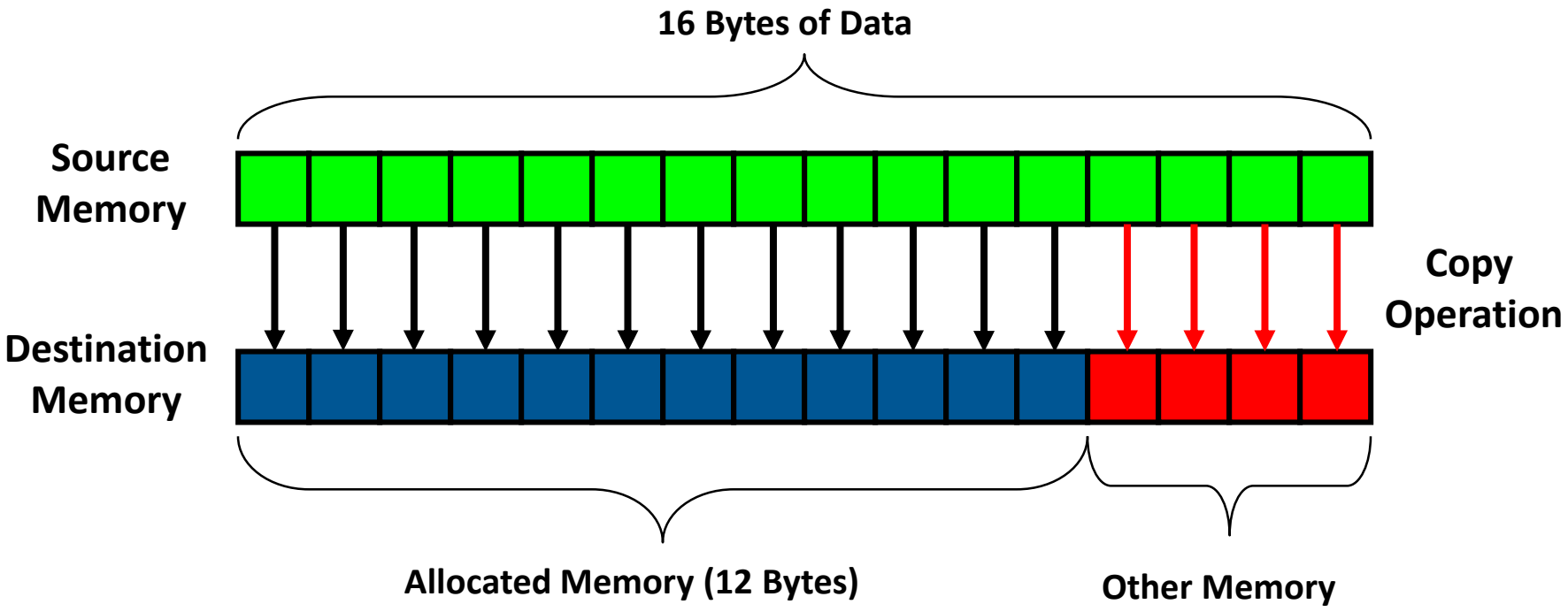
Buffer Overflow

Rank	Score	ID	Name
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

What Is a Buffer Overflow?

A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure.



Buffer Overflows

Are caused when buffer boundaries are neglected and unchecked

Can occur in any memory segment

Can be exploited to modify a

- variable
- data pointer
- function pointer
- return address on the stack

Smashing the Stack for Fun and Profit (Aleph One, *Phrack 49-14*, 1996) provides the classic description of buffer overflows.

Smashing the Stack

Occurs when a buffer overflow overwrites data in the memory allocated to the execution stack.

Successful exploits can overwrite the return address on the stack, allowing execution of arbitrary code on the targeted machine.

This is an important class of vulnerability because of the

- occurrence frequency
- potential consequences

Vulnerable Code 1

```
printf("Enter a new first name.\n");
printf(">>> ");
rc = scanf("%s", records[idx].first_name);
if (rc != 1) {
  printf("Invalid input.\n");
  exit(1);
}
```

Vulnerable Code 2

```
printf("Enter a new first name.\n");
printf(">>> ");
rc = scanf("%s", records[idx].first_name);
if (rc != 1) {
  printf("Invalid input.\n");
  exit(1);
}
```

Reads until space or input exhausted

Completely oblivious to end of buffer!

Mitigation

```
void* rp;
...
printf("Enter a new first name.\n");
printf(">>> ");
rp = fgets(records[idx].first_name,
          sizeof(records[idx].first_name), stdin);
  if(rp == NULL) {
  printf("Invalid input  n");
  exit(1);
}
```

Reads until input exhausted or buffer limit reached

Buffer Overflow Summary

Sometimes the easier code is the less secure code.

- Many C standard library functions do no bounds check
  - Use functions that check bounds (e.g., **`fgets()`**)
- Can write outside array bounds without library function
  - Make sure all array indexes & pointer arithmetic are within range!

Not possible in memory-safe languages like Java and Perl

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Concurrency

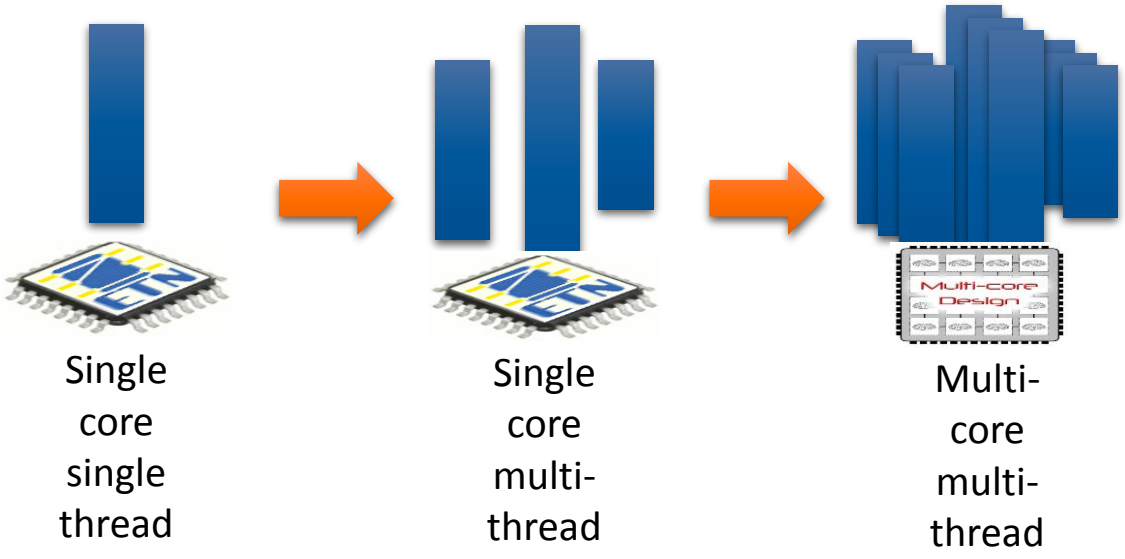
Concurrency

Concurrency occurs when two or more separate execution flows are able to run simultaneously [Dijkstra 65].

Examples of independent execution flows include

- threads
- processes
- tasks

Concurrent execution of multiple flows of execution is an essential part of a modern computing environment.



Race Conditions

A race condition is an execution ordering of concurrent flows that results in undesired behavior.

Race conditions are possible in runtime environments, including operating systems, that must control access to shared resources, especially through process scheduling.

- Race conditions are a frequent source of vulnerabilities.
- Race conditions are particularly insidious because they are non-deterministic or timing dependent.
  - difficult to detect, reproduce, and eliminate
  - can cause errors such as data corruption or crashes [Amarasinghe 2007].

Deadlock

Deadlock occurs whenever two or more control flows block each other in such a way that none can continue to execute.

Deadlock results from a cycle of concurrent execution flows in which each flow in the cycle has acquired a synchronization object that results in the suspension of the subsequent flow in the cycle.

Deadlock can result in a denial-of-service attack.

- VU#132110 Apache HTTP Server versions 2.0.48 and prior contain a race condition in the handling of short-lived connections.
- When using multiple listening sockets, a short-lived connection on a rarely-used socket may cause the child process to hold the accept mutex, blocking new connections from being served until another connection uses the socket.

Concurrency Summary

Concurrency is hard.

- Bugs difficult to reproduce
- Lack of suitable test & debug tools
- Standardization is late and underspecified
- Lack of analysis tools

Main benefit is improved performance (limited by parallelization quotient & Amdahl's Law)

When building a concurrent application

- Make sure improved performance is worth the hassle.
- Establish a good simple design and enforce it!

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Privilege Escalation

Privilege System

Integrated with a larger system

Delegation of authority

Java privilege system

Grants different privileges to different code segments in the same program

Other examples:

- UNIX privileges and permissions
- Windows NT-based privileges
- Android Permission System

Well-Behaved Applets

Java applets run in a security sandbox

- Chaperoned by a **`SecurityManager`**, which throws a **`SecurityException`** if applet tries to do anything forbidden

Sandbox prevents applets from

- Accessing the file system
- Accessing the network
  - EXCEPT the host it came from
- Running external programs
- Modifying the security manager

A signed applet may request privilege to do these things.

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

Java Applet Demo

Java Applet Exploit

CVE 2012-0507

Rank	Score	ID	Name
		CWE-502	Deserialization of Untrusted Data

A4-Insecure Direct Object References

OBJ03-J. Prevent heap pollution
OBJ06-J. Defensively copy mutable inputs and mutable internal components
SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

Trojan BackDoor.Flashback

Malware targeting Mac OS X

First discovered by Intego in September 2011

- Did not use Java then, mimicked Flash installer

Modified to use Java vul in March 2012

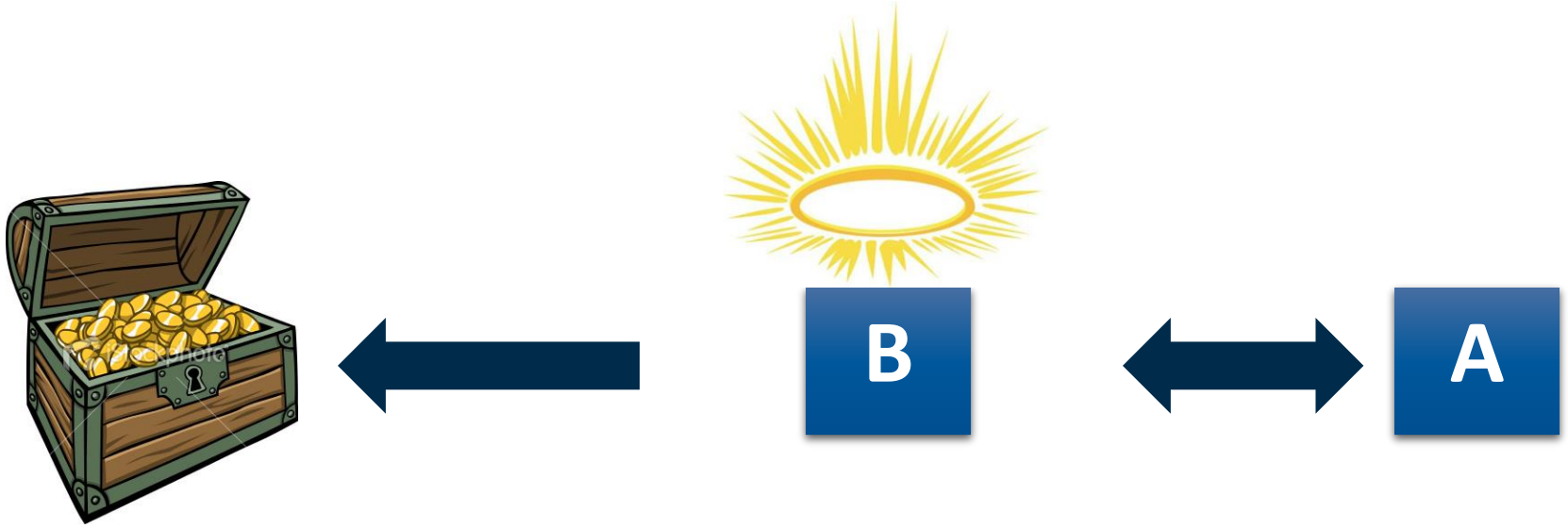
- Oracle had already released Java patch.
  - But Apple hadn't applied it!

Botnet of 600,000 infected Macs

- according to Dr.WEB® Anti-virus

22,000 Macs still infected as of January 2014.

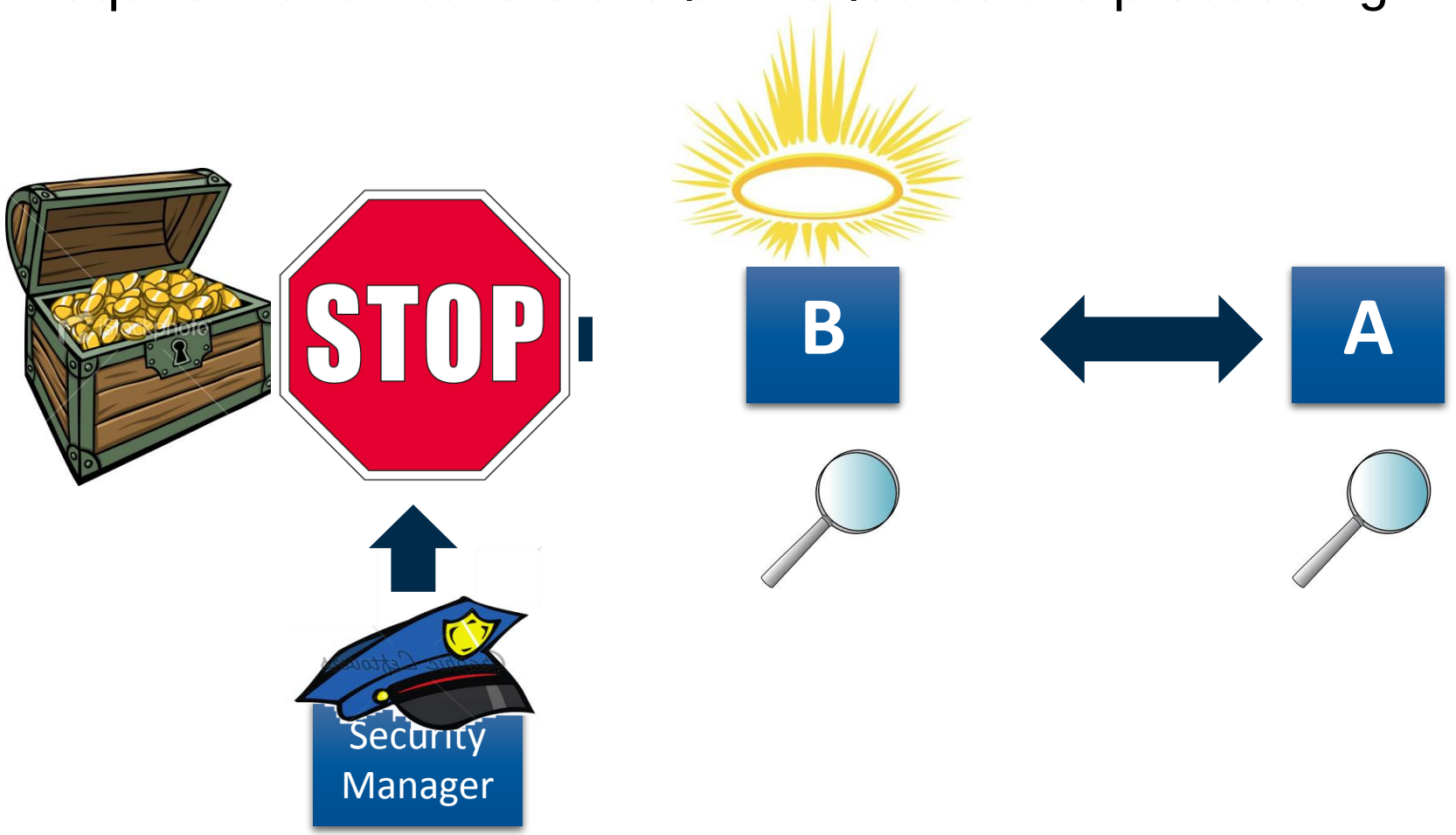
Confused Deputy Problem [1]



Q: If class A is unprivileged and class B is privileged, how do we make sure that class A doesn't trick class B into doing something privileged on A's behalf?

Confused Deputy Problem 2

A: Require that all callers are privileged before proceeding.



Java Applet Summary

Even popular software like Java is insecure!

Privileged code is a more lucrative target than unprivileged code!

- Vulnerabilties more costly

Beware obscure features!

- They may be more useful to attackers than you.

Beware Confused Deputy

Language-independent

Secure Coding Tutorial

Introduction

Injection

- SQL
- OS Command
- Format String

Memory Corruption

- Buffer Overflow

Concurrency

Privilege Escalation

- Java Applet

Conclusion

Secure Coding Tutorial

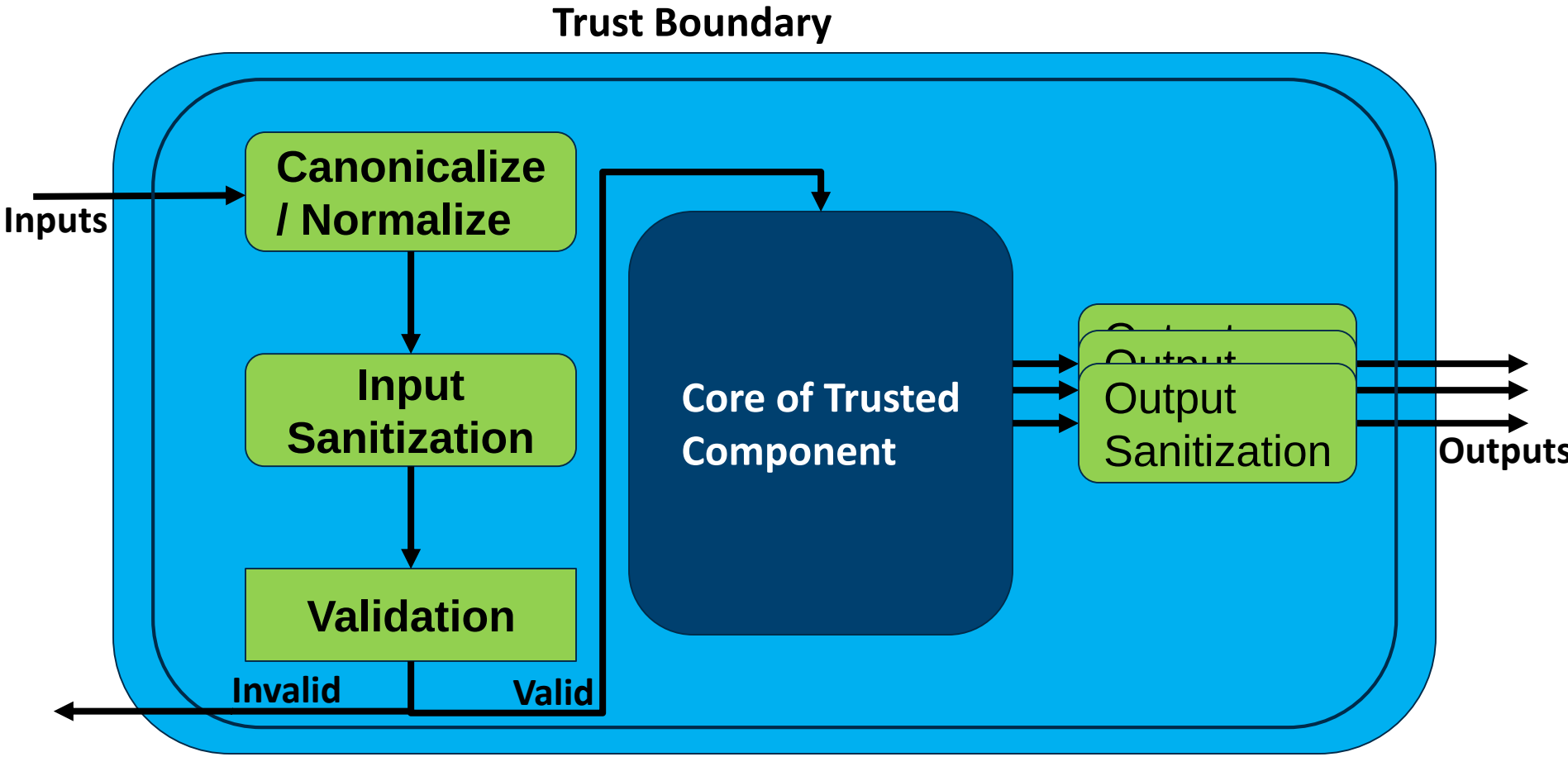
Conclusion

2011 CWE/SANS Top 25 Most Dangerous Software Errors

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[23]	61.0	CWE-134	Uncontrolled Format String
		CWE-502	Deserialization of Untrusted Data

http://cwe.mitre.org/top25/#Listing

Trust Boundary Diagram



Injection Summary

Sanitize any untrusted input that goes to a subsystem

- Databases
- OS Commands
- Format strings
- Web browser (HTML)
- Etc.

Use sanitization provided by your platform, if possible

- Java's **`PreparedStatement`**
- **Perl's 3-argument `open()`**
- **POSIX's `realpath()`**

Don't worry if your sanitization invalidates the input.

- Butchered input less egregious than injection!

Language-independent

Complexity Summary

Beware obscure features!

- They may be more useful to attackers than you.

Layers of abstraction help to obscure features.

- Perl's file-open mechanism buried under **`<>, -n, -p`**
- Many C library functions do not prevent buffer overflow.
  - Buffer overflow also possible w/o library functions
- Quirks in Java's
  - **`SecurityManager`**
  - **`ClassLoader`**
  - Deserialization

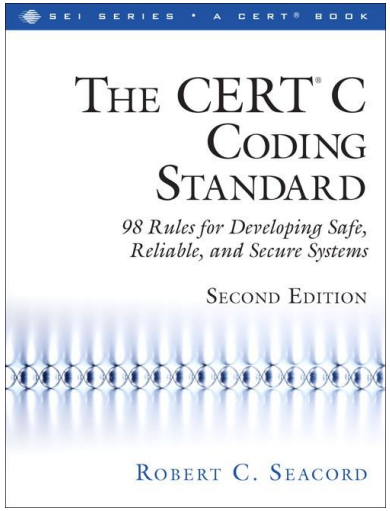
Memory Safety

Use a memory-safe language. (Java, Perl, others)

OR

Check all your:

- Array accesses
- Pointer arithmetic
- Memory allocation & deallocation



Prevent:

- Undefined Behavior
  - C11 lists 203 cases of explicit undefined behavior.
- CERT rule violations
  - The SEI CERT C Coding Standard lists 98 rules.

Misc. Summary

- Even popular software like PHP is insecure!
- Sometimes the easier code is the less secure code.
- It is cheaper to prevent vulnerabilities during development.
- Stay up-to-date with patches.
  - And make sure your platform does too.

For More Information

Contact Presenter

David Svoboda

svoboda@cert.org

+1 412.268.3965

Contact CERT:

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue

Pittsburgh PA 15213-3890

USA

Visit CERT® websites:

http://www.cert.org/secure-coding

https://www.securecoding.cert.org

The End

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Secure Coding Tutorial

HTML Injection (XSS) Demo

Cross-Site Scripting

Rank	Score	ID	Name
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

A3-Cross-Site Scripting (XSS)

IDS33-PL. Sanitize untrusted data passed across a trust boundary

Vulnerable Code (Java)

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws IOException, ServletException {
  response.setContentType("text/html;charset=UTF-8");
  try (PrintWriter out = response.getWriter()) {
    // ...
    String userName = request.getParameter("userName");
    if (userName == null) {
      // ...
    } else {
      out.println("It is a pleasure to meet you, ");
      // Deletes non-character code points
      out.println(userName.replaceAll("[\\p{Cn}]", ""));
      out.println("!");
...
```

Vulnerable Code (Java)

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws IOException, ServletException {
  response.setContentType("text/html;charset=UTF-8");
  try (PrintWriter ou                    ter()) {
    // ...
    String userName =                    er("userName");
    if (userName == null
      // ...
    } else {
      out.println("It is a pleasure to meet you, ");
      // Deletes non-character code points
      out.println(userName.replaceAll("[\\p{Cn}]", ""));
      out.println("!");
...
```

Normalizes input, but does not sanitize it.

Mitigation

```
private String sanitize(String s) {
  // Deletes non-character code points
  s = s.replaceAll("[\\p{Cn}]", "");
  // Replace anything that is not alphanumeric
  s = s.replaceAll("[^A-Za-z_]", "_");
  return s;
}
```

Sanitization is done after normalization!

```
      out.println("It is a pleasure to meet you, ");
      // Deletes non-character code points
      out.println(sanitize(userName));
      out.println("!");
...
```

HTML Injection (XSS) Summary

Know your trust boundaries

- Can untrusted users access your website?

Sanitize your website's input!

- Prevent users from entering
  - Images
  - Other HTML Tags
  - Javascript

Don't worry if your sanitization invalidates the input.

- Butchered input less egregious than XSS

Don't forget to sanitize your website's output, too!

- Purge sensitive information

Language-independent (not specific to Java)

Perl

The summary is that when Perl opens files using <>, it uses Perl's open syntax. And when you tell Perl to open a file that ends with |, it treats it as a shell command to send input to, and executes it!

Buffer Overflow Vulnerable Code

```
printf("Enter a new last name.\n");
printf(">>> ");
rc = scanf("%s", records[idx].last_name);
if(rc != 1) {
      printf("Invalid input.\n");
      exit(1);
}
```

Key Ideas: Privilege Separation & Privilege Minimization

Privilege Separation

- A system is decomposed into separate components
- Each component possesses *only* those privileges required for it to function
- Consequence: component cannot perform *other* privileged operations
  - Limits impact of errors and of successful attacks

Privilege Minimization

- Privileges are *disabled* most of the time
- Privileges are enabled exactly and only when required
- Consequences:
  - Reduces amount of privileged code
    - Easier to get it right
    - Reduces cost of review
  - Temporally limits certain attack opportunities

Key Idea: Distrustful Decomposition

Components have limited trust in each other

- Similar to compartmentalized security

Consequence: Must manage interactions between components with care

- Canonicalize, Sanitize, Normalize & Validate inputs
  - Goal: Limit potential attacks
- Sanitize outputs
  - Goal: Prevent information and capability leaks
- Addressed by rules shown later