Secure Coding Practices (and Other Good Things)

James A. Kupsch

Barton P. Miller

Computer Sciences Department
University of Wisconsin

kupsch@cs.wisc.edu
bart@cs.wisc.edu

Elisa Heymann

Computer Architecture and
Operating Systems Department
Universitat Autònoma de Barcelona

elisa@cs.wisc.edu

The 4th DIRAC User Workshop
CERN
May 28, 2014

Who we are

Bart Miller
Jim Kupsch
Vamshi Basupalli
Josef Burger

Elisa Heymann
Eduardo Cesar
Manuel Brugnoli
Max Frydman

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

http://www.cs.wisc.edu/mist/papers/VAshort.pdf

Our experience

Condor, University of Wisconsin
Batch queuing 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++

Our experience

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 Autònoma de Barcelona
Resource Mgr for Parallel & Interactive Applications
4 vulnerabilities — 97 KLOC of C++

ARGUS 1.2, HIP, INFN, NIKHEF, SWITCH
gLite Authorization Service
0 vulnerabilities — 42 KLOC of Java and C

Our experience

VOMS Core INFN
Virtual Organization Management System
1 vulnerability 161 KLOC of Bourne Shell, C++ and C

iRODS, DICE
Data-management System
9 vulnerabilities 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.

The Art of Software Security Assessment

- **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

The attack can come from a

- **program or script**
- **human with interactive access**

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.

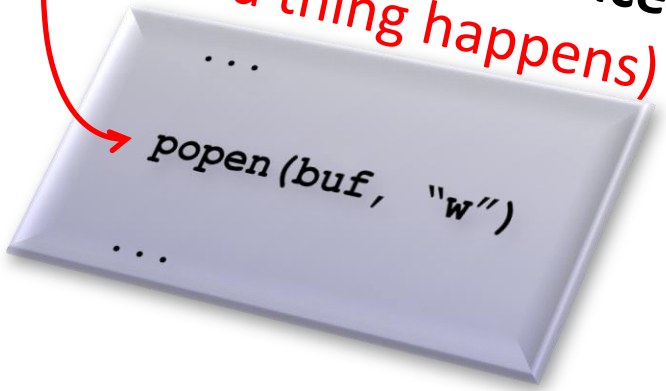
- Start at the user's entry point to the program: the attack surface:
  - Network input buffer
  - Field in a form
  - Line in an input file
  - Environment variable
  - Program option
  - Entry in a database
  - …
- Attack surface: the set of points in the program's interface that can be controlled by the user.

The Path of an Attack

The Attack Surface
(user data enters here)

```
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);
```

The Impact Surface
(bad thing happens)



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

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.

Roadmap

- Introduction
- Pointers and Strings
- Numeric Errors
- Race Conditions
- Exceptions
- Privilege, Sandboxing and Environment
- Injection Attacks
- Web Attacks

Discussion of the Practices

- Description of vulnerability
- Signs of presence in the code
- Mitigations
- Safer alternatives

Pointers and Strings

Buffer Overflows

http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.html#Listing

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

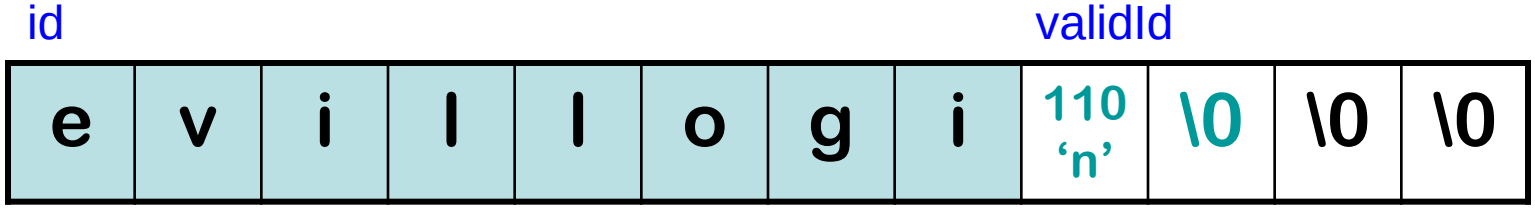
- An overflow overwrites memory adjacent to a buffer
- This memory could be
  - Unused
  - Code
  - Program data that can affect operations
  - Internal data used by the runtime system
- Common result is a crash
- Specially crafted values can be used for an attack

Buffer Overflow of User Data Affecting Flow of Control

C/C++

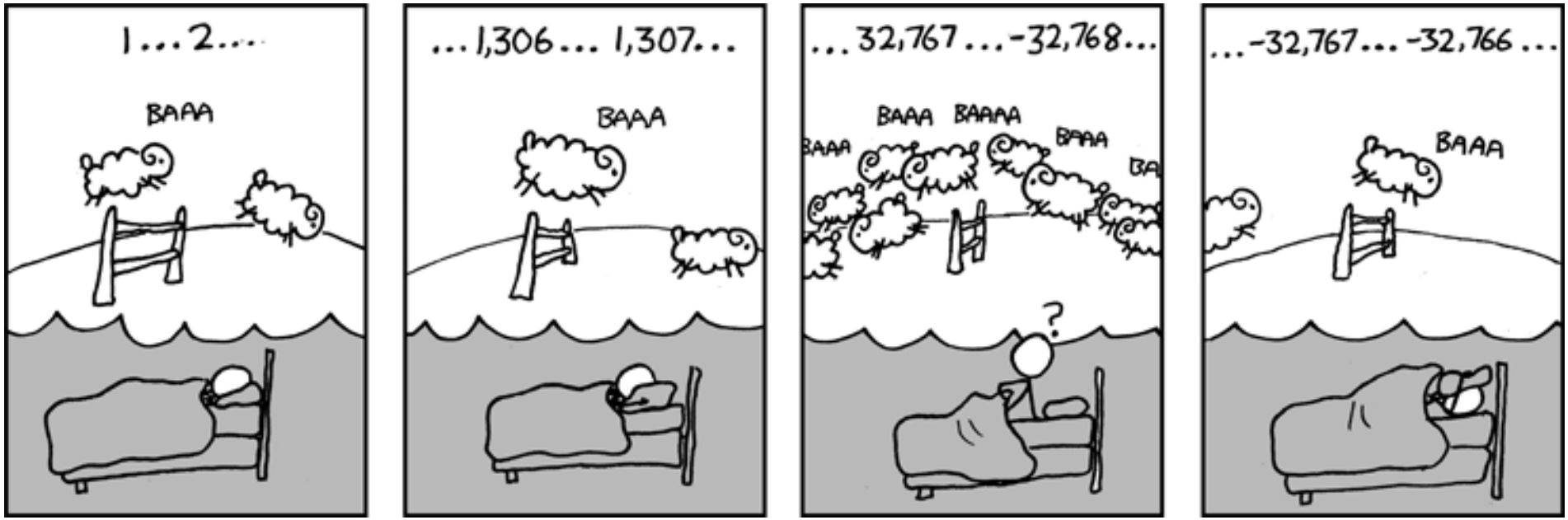
```
char id[8];
int  validId = 0;    /* not valid */
```


```
gets(id);       /* reads "evillogin"*/
```




```
/* validId is now 110 decimal */
if (IsValid(id))          /* <- false */
    validId = 1;              /* not executed */
if (validId)              /* <- true */
    DoPrivilegedOp();         /* is executed */
```

Numeric Errors



http://xkcd.com/571

Integer Vulnerabilities

- Description
  - In many programming languages (C, C++, Java, Perl, Python 2.x have problems; Python 3.x is OK), integers are module 2^n, allow silent unexpected results
    - 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

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

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.

Successful Race Condition Attack

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

time

Thread 1	Thread 2	Balances Bob	Balances Ian
XfrFunds(Bob, Ian, 100)	XfrFunds(Bob, Ian, 100)	100	0
srcAmt = 100			
	srcAmt = 100		
srcAmt - 100 < 0 ?			
	srcAmt - 100 < 0 ?		
srcAcct.setBal(100 - 100)		0	
	srcAcct.setBal(100 - 100)	0	
dst.setBal(0 + 100)			100
	dst.setBal(0 + 100)		200

Mitigated Race Condition Attack

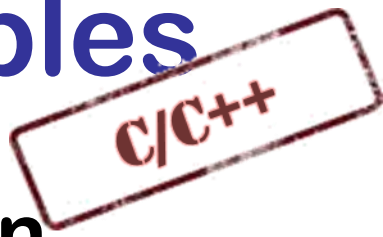
```
public void TransFunds(Account srcAcct, Account dstAcct, int xfrAmt)
{
      if (xfrAmt < 0)  FatalError();
      synchronized(srcAcct) {
         int srcAmt = srcAcct.getBal();
         if (srcAmt - xfrAmt < 0)
            FatalError();
         srcAcct.setBal(srcAmt - xfrAmt);
      }
      synchronized(dstAcct)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

Thread 1	time	Thread 2	Bob	Ian
XfrFunds(Bob, Ian, 100)		XfrFunds(Bob, Ian, 100)	100	0
In use srcAcct? No, proceed.				
		In use srcAcct? Yes, wait.		
srcAmt = 100				
srcAmt – 100 < 0 ?				
srcAcct.setBal(100 – 100)			0	
In use dstAcct? No, proceed.		srcAmt = 0		
dst.setBal(0 + 100)		srcAmt – 100 < 0? Yes, fail		100

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 only once
    - 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
    - http://www.cs.wisc.edu/mist/safefile

James A. Kupsch and Barton P. Miller, "How to Open a File and Not Get Hacked," *2008 Third International Conference on Availability, Reliability and Security* (ARES), Barcelona, Spain, March 2008.

Race Condition Examples

C/C++

- Your Actions

```
s=strdup("/tmp/zXXXXXX")
tempnam(s)
// s now "/tmp/zRANDOM"


f = fopen(s, "w+")
// writes now update
// /etc/passwd
```

time

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++.**

```
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

Exception Suppression

1. User sends malicious data

```
user="admin",pwd=null
```

```
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;
}
```

2. System grants access

```
Login() returns true
```

Unusual or Exceptional Conditions Mitigation

JAVA

1. User sends malicious data | `user="admin",pwd=null`

```
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;
}
```

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.\n");
    print(e + "\n");
    e.printStackTrace();
    return;
  }
}
```

```
void ValidatePwd(… user, … pwd)
        throws BadUser, BadPwd  {
  realPwd = GetPwdFromDb(user);
  if (realPwd == null)
    throw BadUser("user=" + user);
  if (!pwd.equals(realPwd))
    throw BadPwd("user=" + user
          + " pwd=" + pwd
          + " expected=" + realPwd);
…
```

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)

The Right Amount of Information


```
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");
       return;
   }
}
```

Log sensitive information

Generic error message
(id links sensitive information)

```
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));
 …
}
```

User and password are hashed
(minimizes damage if breached)

Privilege, Sandboxing, and Environment

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
  - http://www.cs.wisc.edu/mist/safefile
  - 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*>

<*user-data*>	Directory Movement
../	up
./ or empty string	none
<dir>/	down

- 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

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

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
  - No user control of 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**

```
$u = " '; drop table t --";
$sth = $dbh->do("select * from t where u = '$u'");
```

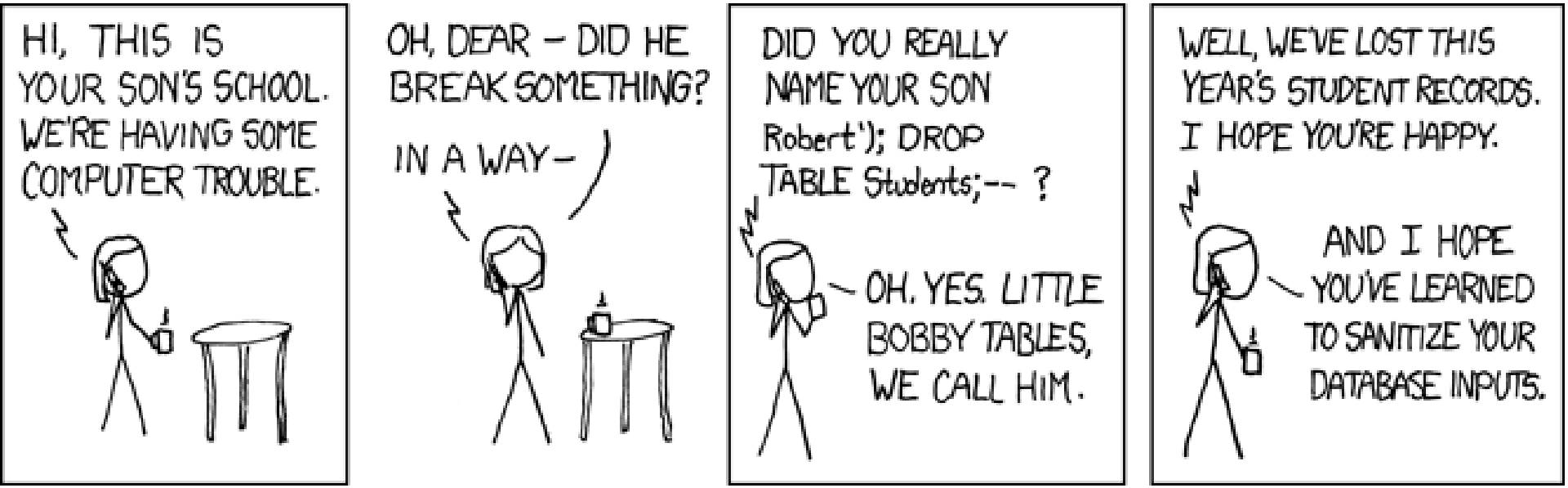
Database sees <u>two</u> statements:

```
select * from t where u = ' '; drop table t --'
```

- **Use *prepared statements* to mitigate**

```
$sth = $dbh->do("select * from t where u = ?", $u);
```

  - SQL statement template and value sent to database
  - No mismatch between intention and use



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

Perl Command Injection Examples

- `open(CMD, "|/bin/mail -s $sub $to");`
  - Unsafe if `$to` is `"badguy@evil.com; rm -rf /"`
- `open(CMD, “|/bin/mail -s '$sub' '$to'");`
  - Unsafe 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

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

JAVA

1. User sends malicious data

```
hostname="x.com;rm -rf /*"
```

2. Application uses nslookup to get DNS records

```
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

JAVA

1. User sends malicious data

```
hostname="x.com;rm -rf /*"
```

2. Application uses nslookup only if input validates

```
String rDomainName(String hostname)  {
  …
  if (hostname.matches("[A-Za-z][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"`

Code Injection

- Cause
  - Program generates source code from template
  - User supplied data is injected in template
  - Failure to neutralized 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. logfile – name's value is user controlled

```
name = John Smith
name = ');import os;os.system('evilprog');#
```

Read logfile

2. Perl log processing code – uses Python to do real work

```
%data = ReadLogFile('logfile');
PH = open("|/usr/bin/python");
print PH "import LogIt\n";w
while (($k, $v) = (each %data)) {
  if ($k eq 'name')  {
  print PH "LogIt.Name('$v')";
}
```

Start Python, program sent on stdin

3. Python source executed – 2nd LogIt executes arbitrary code

```
import LogIt;
LogIt.Name('John Smith')
LogIt.Name('');import os;os.system('evilprog');#')
```

Code Injection Mitigated

1. logfile – name's value is user controlled

```
name = John Smith
name = ');import os;os.system('evilprog');#
```

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

```
%data = ReadLogFile('logfile');
PH = open("|/usr/bin/python");
print PH "import LogIt\n";w
while (($k, $v) = (each %data)) {
  if ($k eq 'name')  {
    $q = QuotePyString($v);
    print PH "LogIt.Name($q)";
}
```

```
sub QuotePyString  {
  my $s = shift;
  $s =~ s/\\/\\\\/g;     # \  ➔ \\
  $s =~ s/'/\\'/g;       # '  ➔ \'
  $s =~ s/\n/\\n/g;      # NL ➔ \n
  return "'$s'";         # add quotes
}
```

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

```
import LogIt;
LogIt.Name('John Smith')
LogIt.Name('\');import os;os.system(\'evilprog\');#')
```

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)

3. Generated HTML displayed by browser

```
<html>
…
You searched for:
widget
…
</html>
```

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:\n" + query);
}
…
```

Reflected Cross Site Scripting (XSS)

JAVA

3. Generated HTML displayed by browser

```
<html>
...
You searched for:
<script>alert('Boo!')</script>
...
</html>
```

1. Browser sends request to web server

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

2. Web server code handles request

```
...
String query = request.getParameter("q");
if (query != null) {
    out.writeln("You searched for:\n" + query);
}
...
```

XSS Mitigation

JAVA

3. Generated HTML displayed by browser

```
<html>
...
Invalid query
...
</html>
```

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:\n" + query);
    }  else  {
        out.writeln("Invalid query");
    }
}
...
```

Cross Site Request Forgery (CSRF)

- CSRF is when loading a web pages 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)
  - http://bank.com/xfer?amt=1000&toAcct=joe HTTP GET method
  - HTTP POST method

```
<form action=/xfer method=POST>
  <input type=text name=amt>
  <input type=text name=toAcct>
</form>
```

- 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
   - Bad guy's server
   - Fake 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
```

2. evil.com returns HTML

```
<html>
…
<img src='http://bank.com/xfer?amt=1000&toAcct=evil37'>
…
</html>
```

3. Browser sends attack

```
http://bank.com/xfer?amt=1000&toAcct=evil37
```

4. bank.com server code handles request

```
…
String id = response.getCookie("user");
userAcct = GetAcct(id);
If (userAcct != null)  {
    deposits.xfer(userAcct, toAcct, amount);
}
```

CSRF Mitigation

1. User visits evil.com

2. evil.com returns HTML

3. Browser sends attack

JAVA

Very unlikely attacker will provide correct nonce

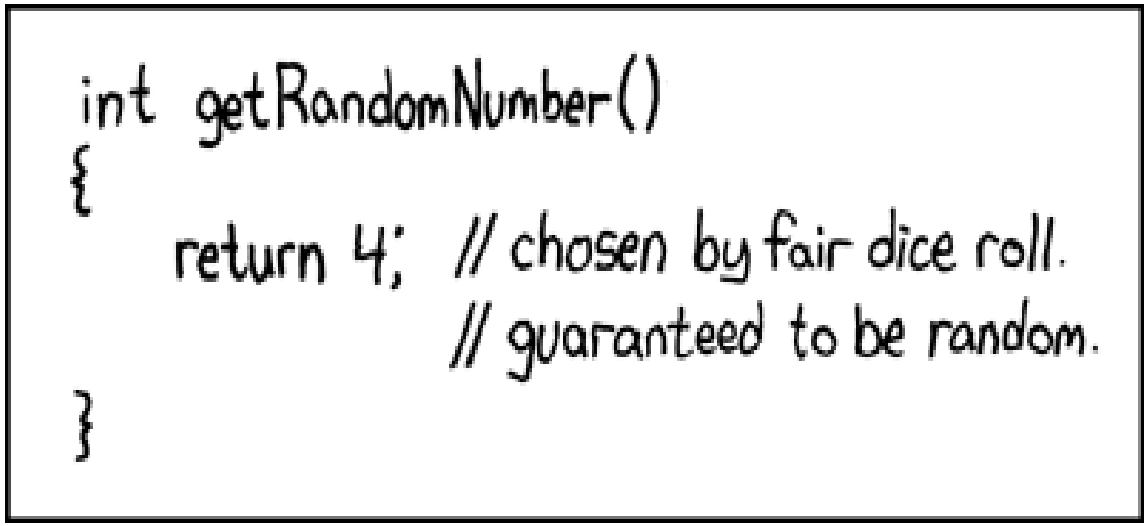
4. bank.com server code correctly handles request

```
...
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);
}
```

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



http://xkcd.com/221

- **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

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

1. Tricks user to visit

```
http://bank.com/login;JSESSIONID=123
```

2. User Logs In

```
http://bank.com/login;JSESSIONID=123
```

3. Creates the session

```
HTTP/1.1 200 OK
Set-Cookie:
JSESSIONID=123
```

4. Impersonates the user

```
http://bank.com/home
Cookie: JSESSIONID=123
```

```
if(HttpServletRequest.getRequestedSessionId() == null)
{
      HttpServletRequest.getSession(true);
}
...
```

Mitigated Hijacking Attack

1. Tricks user to visit

```
http://bank.com/login;JSESSIONID=123
```

2. User Logs In

```
http://bank.com/login;JSESSIONID=123
```

3. Creates the session

```
HTTP/1.1 200 OK
Set-Cookie:
JSESSIONID=XXX
```

4. Impersonates the user

```
http://bank.com/home
Cookie: JSESSIONID=123
```

```
HttpServletRequest.invalidate();
HttpServletRequest.getSession(true);
...
```

Secure Coding Practices (and Other Good Things)

James A. Kupsch
Barton P. Miller
{kupsch,bart}@cs.wisc.edu

Elisa Heymann
Elisa.Heymann@uab.es

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

http://www.cs.wisc.edu/mist/papers/VAshort.pdf