Vulnerability Assessment and Secure Coding Practices for Middleware

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

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

Epic Systems
March 11, 2010

Part 2 Roadmap

• Part 1: Vulnerability assessment process
  – Buffer overflows
  – Injections
  – Integer
  – Race conditions
  – Privileges
  – Command line
  – Environment
  – Denial of service
  – General engineering
  – Compiler warnings

• Part 2: Secure coding practices
  – Introduction
  – Handling errors
  – Numeric parsing
  – Missing error detection
  – ISO/IEC 24731
  – Variadic functions
Discussion of the Practices

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

Handling Errors

• If a call can fail, always check for errors
  optimistic error handling (i.e. none) is bad
• Error handling strategies:
  – Handle locally and continue
  – Cleanup and propagate the error
  – Exit the application
• All APIs you use or develop, that can fail, must be able to report errors to the caller
• Using exceptions forces error handling
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`, `strtoll`, `strtoull`, `strtof`, `strtol`, `strtold` which allow error detection

Missing Error Detection

- `strcat`, `strcpy`, `strncat`, `strncpy`, `gets`, `getpass`, `getwd`, `scanf` (using `%s` or `% [...] without width specified)
  - Never use these
  - Unable to report if buffer would overflow (not enough information present)
  - Safer alternatives exist
ISO/IEC 24731
Extensions for the C library:
Part 1, Bounds Checking Interface

• Functions to make the C library safer
• Meant to easily replace existing library calls with little or no other changes
• Aborts on error or optionally reports error
• Very few unspecified behaviors
• All updated buffers require a size param
• http://www.open-std.org/jtcl/sc22/wg14

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 */

id validId
\0 \0 \0 \0
gets(id); /* reads "evillogin"*/

id validId
\0 \0 \0 \0

/* 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>Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>gets(s)</td>
<td>fgets(s, sLen, stdin)</td>
</tr>
<tr>
<td>getcwd(s)</td>
<td>getwd(s, sLen)</td>
</tr>
<tr>
<td>scanf(&quot;%s&quot;, s)</td>
<td>scanf(&quot;%100s&quot;, s)</td>
</tr>
</tbody>
</table>

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 the function

Proper usage: concat s1, s2 into dst

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** \(n \geq (dst\text{Size} - \text{strlen}(dst))\)

- **strncpy** \((dst, src, n)\)
  - Writes \(n\) chars into \(dst\), if \(\text{strlen}(src) < n\), it fills the other \(n - \text{strlen}(src)\) chars with 0’s
  - If \(\text{strlen}(src) \geq n\), \(dst\) 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** \((buf, buf\text{Size}, fmt, \ldots)\) and **vsnprintf**
  - Returns number of bytes, **not** including \(\text{\textbackslash 0}\) that **would**’ve been written if was enough room.
  - Truncation detection possible
    - (result \(\geq\) buf\text{Size} implies truncation)
  - Use as safer version of **strcpy** and **strcat**

Proper usage: concat s1, s2 into dst

```c
r = snprintf(dst, dst\text{Size}, "%s%s", s1, s2);
If (r \geq\) dst\text{Size})
    \{ERROR("truncation");\}
```
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 quoting 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 2 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
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

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)
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

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
    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

Race Condition Temporary Files

• Temporary directory (/tmp) is a dangerous area of the file system
  – Any process can create a directory entry there
  – Usually has the sticky bit set, so only the owner can delete their files

• Ok to create true temporary files in /tmp
  – Create using mkstemp, unlink, access through returned file descriptor
  – Storage vanishes when file descriptor is closed

• Safe use of /tmp directory
  – create a secure directory in /tmp
  – use it to store files
Race Condition Examples

- **Your Actions**
  ```c
  s = strdup("/tmp/zXXXXXX")
  tempnam(s)
  // s now "/tmp/zRANDOM"
  
  f = fopen(s, "w+")
  // writes now update
  // /etc/passwd
  ```

- **Attackers Action**
  ```c
  link = "/etc/passwd"
  file = "/tmp/zRANDOM"
  symlink(link, file)
  ```

Safe Version
```c
fd = mkstemp(s)
// writes now update
// /etc/passwd
```  

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`
- Need to `chdir("/")` to somewhere underneath the new root directory, otherwise relative pathnames are not restricted
- Need to recreate all support files used by program in new root: `/etc`, libraries, ...

Insecure Permissions

- Set `umask` when using `mkstemp` or `fopen`
  - File permissions need to be secure from creation to destruction
- Don’t write sensitive information into insecure locations (directories need to have restricted permission to prevent replacing files)
- Executables, libraries, configuration, data and log files need to be write protected
Insecure Permissions

• If a file controls what can be run as a privileged, users that can update the file are equivalent to the privileged user

File should be:
  – Owned by privileged user, or
  – Owned by administrative account
    • No login
    • Never executes anything, just owns files

• DBMS accounts should be granted minimal privileges for their task

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
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
  – Avoid PATH, LD_LIBRARY_PATH, and other variables that are directory lists used to look for execs and libs
General Software Engineering

• Don’t trust user data
  – You don’t know where that data has been
• Don’t trust your own client software either
  – It may have been modified, so always revalidate data at the server.
• Don’t trust your operational configuration either
  – If your program can test for unsafe conditions, do so and quit
• Don’t trust your own code either
  – Program defensively with checks in high and low level functions
• KISS - Keep it simple, stupid
  – Complexity kills security, its hard enough assessing simple code

Let the Compiler Help

• Turn on compiler warnings and fix problems
• Easy to do on new code
• Time consuming, but useful on old code
• Use lint, multiple compilers
• -Wall is not enough!
  gcc: -Wall, -W, -O2, -Werror, -Wshadow,
  -Wpointer-arith, -Wconversion, -Wcast-qual,
  -Wwrite-strings, -Wunreachable-code and many more
  – Many useful warning including security related
    warnings such as format strings and integers
Books


Questions?

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