Vulnerability Assessment and Secure Coding Practices for Middleware

Part 2

James A. Kupsch
Computer Sciences Department
University of Wisconsin

Part 2 Roadmap

- Part 1: Vulnerability assessment process
- Part 2: Secure coding practices
  - Introduction
  - Handling errors
  - Numeric parsing
  - ISO/IEC 24731 intro
  - Variadic functions
  - Buffer overflows
  - Injections
  - Directory traversal
  - Integer
  - Race conditions
  - File system issues
  - Canonical form
  - Privileges
  - Command line
  - Environment
  - Denial of service
  - Information leaks
  - Memory allocators
  - General engineering
  - Compiler warnings

Vulnerability Types

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

Handling Errors

- If a call can fail, always check the status
- When an error is detected
  - Handle locally and continue
  - Cleanup and propagate the error
  - Exit the application
- All APIs you use, or develop, that can fail need to be able to report errors to the caller
- Using exceptions makes it harder to ignore

Numeric Parsing No Error Indication

- `atoi`, `atol`, `atof`, `scanf` family (with `%u`, `%i`, `%d`, `%x` and `%o` specifiers)
  - Out of range values results in unspecified behavior
  - Non-numeric input results in 0
  - Use `strtol`, `strtoul`, `strtoll`, `strtoull`, `strtof`, `strtod`, `strtold` which provide an error indication

Correct Numeric Parsing

```
char *endptr;
long longVal;

errno = 0;
longVal = strtol(s, &endptr, 10);
if (errno == ERANGE)
   {ERROR("overflow");}
if (errno != 0)
   {ERROR("other error");}
if (endptr == 0)
   {ERROR("non-numeric");}
if (*endptr != '\0')
   {ERROR("non-numeric at end");}
if (isspace(*s))
   {ERROR("space at beginning");}
```

Correct Numeric Parsing in C++

- iostream inserter's
  - Type safe
  - All errors set stream to failed (test with `!is`)
  - Use `istringstream` to parse a string

```
istringstream is("123 87.32");
is >> i >> f;
if (!is) {ERROR("parse error");
```

- Boost's `lexical_cast<T>(s)`
  - http://www.boost.org
  - Throw's `bad_lexical_cast` exception on failure

THE UNIVERSITY of WISCONSIN MADISON

SDSC

Not enough information to report an error

- `strcat`, `strcpy`, `strncat`, `strncpy`, `gets`, `getpass`, `getwd`, `scanf` (**with** `%s` **or** `%[…]`, **without width specified)**
  - Unable to report an error if buffer would overflow as it does not have enough information
  - Only secure in rare case where files or strings are verified for secure values before use
  - Never use these

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
- Easy to check errors
- Very few unspecified behaviors
- All updated buffers require a size
- http://www.open-std.org/jtcl/sc22/wg14

ISO/IEC 24731: Run-time Constraints

- A run-time constraint is a property of the arguments that must be true at call time
- A violation is handled by callback
  - Set with `set_constraint_handler_s`
  - Default is `abort_handler_s`
    - Violations not allowed affect program
  - `ignore_handler_s`
    - Allows detection and handling of violations locally
  - Define your own callback

ISO/IEC 24731: Common Run-time Constraints

- `rsize_t` parameter type used to indicate the size of the buffer or amount to copy
  - Violation if `size > RSIZE_MAX`
  - Catches large size caused by integer overflow
- Buffer pointers
  - Violation if `NULL`
- `dst` buffer too small for operation
  - Usually a violation (`snprintf` truncates)

ISO/IEC 24731: gets_s

- `gets_s(buf, bufSize)`
- Like `fgets(buf, bufSize, stdin)`, except new-line removed
- Run-time constraint failure if new-line is not found in bufSize characters
- If error
  - Null-terminates buf
  - Reads complete line and discards instead of returning partial line like `fgets`

Variadic Functions

- C functions that can take a variable number of parameters
- Not type safe
  - Types and number know from format string or implicit and sentinel values are used
- Signs: `va_list` `va_start` `va_arg` `va_end` `<stdarg.h>` `<cstdarg>`
- Common variadic functions
  - `printf`, `scanf`, `syslog` families
  - `execl` family
  - open with `O_CREAT` (3rd argument is the mode)

Variadic Function Safety

- `printf`, `scanf`, `syslog` families
  - Never take the format string from the user
  - Use compile time constants for the format string
  - Turn on compile time warning to check arguments against the format string
  - Use C++ iostreams
- Check all calls to `open` with `O_CREAT` includes the 3rd argument for the mode

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 not together 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
  - Program data that can affect operations
  - Internal data used by the runtime system
- Usual sign is a crash
- Specially crafted values can be used for an attack

Buffer Overflow of User Data Affecting Flow of Control

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

id | | | | | | | | logFunc | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | \0 | \0 | \0 | \0 |

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

id | | | | | | | | logFunc | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| e | v | i | l | g | u | y | x | x79 y | \0 | \0 | \0 |

```
  /* validId is now 121 decimal */
  if (IsValid(id)) validId = 1;
  if (validId) {DoPrivilegedOp();} /* runs */
```


THE UNIVERSITY of WISCONSIN MADISON

SDSC

Pointer Attacks

- First, overwrite a pointer
  - In the code
  - In the run-time environment
    - Heap attacks use the pointers usually at the beginning and end of blocks of memory
- Second, cause the pointer to be used
  - Read user controlled data that causes a security violation
  - Write user controlled data that later causes a security violation

Stack Smashing

- This is a buffer overflow of a variable local to a function that corrupts the internal state of the run-time system
- Target of the attack is the value on the stack to jump to when the function completes
- Can result in arbitrary code being executed
- Not trivial, but not impossible either

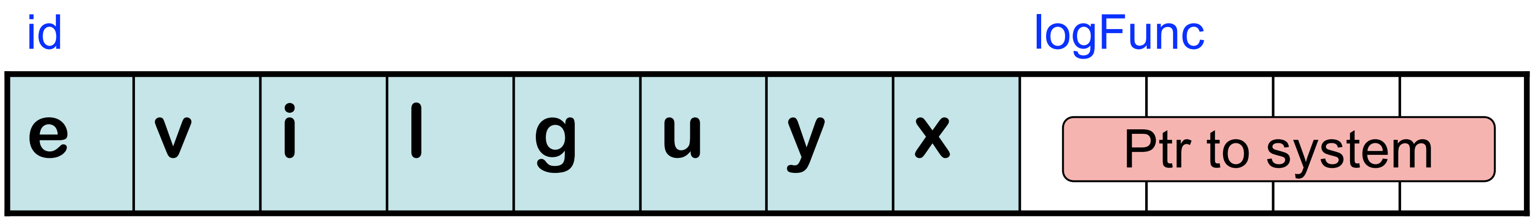
Attacks on Code Pointers

- Stack Smashing is an example
- There are many more pointers to functions or addresses in code
  - Dispatch tables for libraries
  - 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

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


```
  gets(id);        /* reads "evilguyx Ptr to system */
```



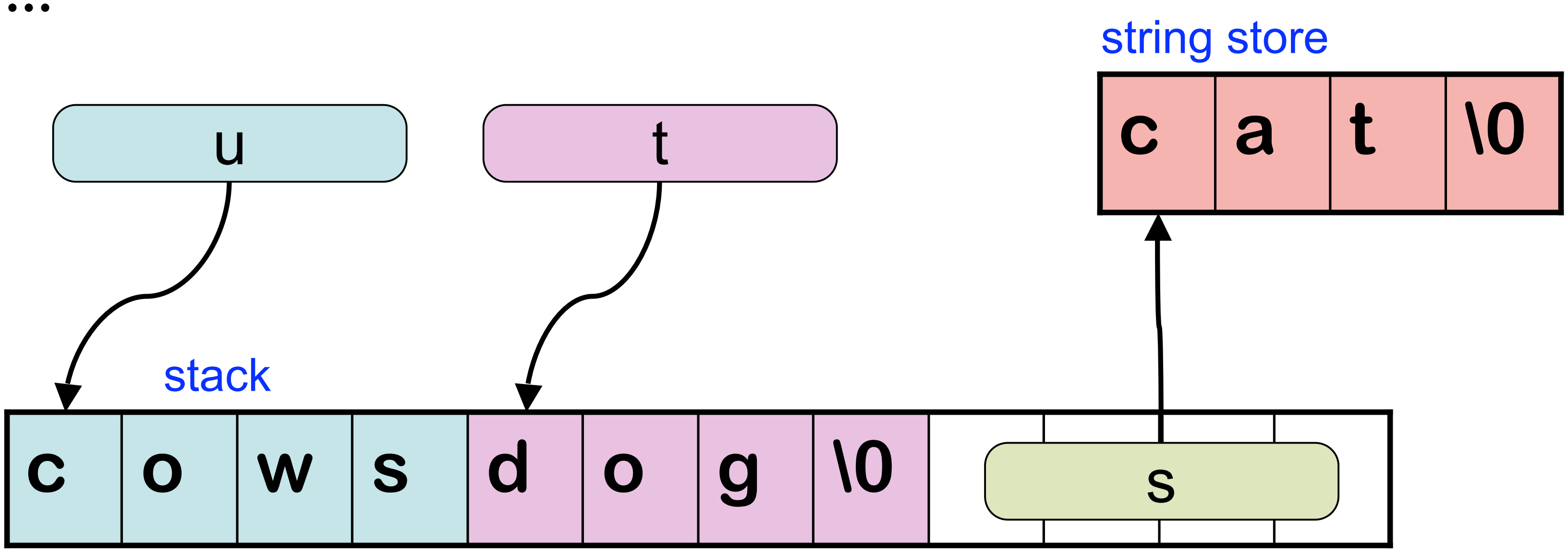

```
  /* equivalent to system(userMsg) */
  logFunc(userMsg);
```

C-style String Design Flaws

- Null terminated array of characters
- Represented by a pointer to this array
- Not a proper type, just a convention
- Only language support is string literals
  - Initialize a char array
  - Create array containing a constant string literal
- Problems
  - Null may be missing
  - Pointers are difficult to use correctly
  - Size of buffer is kept externally from pointer if at all
  - Many common operations are expensive
  - Can't have a string with a null in it

C-style String Example

```
{
  char u[4] = "cows";
  char t[]  = "dog";
  char *s   = "cat";
  …
```



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

Unsafe	Safe
`gets(s)`	`fgets(s, sLen, stdin)`
`getcwd(s)`	`getwd(s, sLen)`
`scanf("%s", s)`	`scanf("%100s", s)`

strcat, strcpy, sprintf, vsprintf

- Destination buffer size not passed
  - Impossible for function to detect overflow
- Difficult to use safely w/o preflight 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), implying *n* must be (dstSize - `strlen(dst)` - 1) or less
- `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
- **Neither allows truncation detection directly from result**

Proper Usage of `strncat` and `strncpy`

- Requires essentially the same check as before
- Checks are inefficient, but required

Proper usage: concat s1, s2 into dst

```
curDstSize = dstSize;
strncpy(dst, s1, curDstSize);
curDstSize -= strlen(s1);
strncat(dst, s2, curDstSize);
curDstSize -= strlen(s2);
If (curDstSize < 1)
      {ERROR("truncation");}
```

Buffer Overflow Danger Signs: `scanf` family

- Max field can not be taken from an argument
  - `*` width suppresses assignment
- `%nc` does not null terminate
- `%ns` and `%n[…]` require a buffer of size *n*+1
- Requires manual coordination of format string, number and types of arguments, and result

Example: 3 items must be coordinated

```
char big[100], small[10];
int r, j;
r = scanf("%99s %9s %d", big, small, &j);
If (r == EOF) ERROR("EOF")
If (r != 3) ERROR("bad line");
```

Unterminated String: `readlink`

- `readlink(path, buf, bufSize)`
- Reads symbolic link referent
- Does not null terminate
- Returns number of characters written to buf or -1 on error

Proper usage: readlink

```
r = readlink(path, buf, bufSize);
If (r == -1)        {ERROR("error in errno");}
If (r == bufSize)   {ERROR("referent truncated");}
buf[r] = '\0';
```

Buffer Overflow Mitigations

- `snprintf(buf, bufSize, fmt, …)` and `vsnprintf`
  - Truncation detection possible (`result >= bufSize` implies truncation)
  - Can be used as a safer version of `strcpy` and `strcat`
  - Officially doesn't exist until C99 standard

Proper usage: concat s1, s2 into dst

```
r = snprintf(dst, dstSize, "%s%s",s1, s2);
If (r >= dstSize)
      {ERROR("truncation");}
```

Safer String Handling: BSD's `strlcpy` and `strlcat`

- `strlcpy(dst, src, size)` and `strlcat(dst, src, size)`
  - `size` is always the size of the `dst` buffer
  - Returns number of chars required
  - `result >= size` indicates truncation
  - dst always null terminated, except `strlcat` where dst is not terminated
  - Can read outside src if not null-terminated
  - Not universally implemented (not in linux)

Safer String Handling:
BSD's strlcpy and strlcat

Proper usage: concat s1, s2 into dst

```
/* safe to just check errors at last call */
(void)strlcpy(dst, s1, dstSize);

r = strlcat(dst, s2, dstSize)
if (r >= dstSize)  {
      if (r == dstSize && dst[r] != '\0')  {
            /* this should not happen as
               strlcpy will always terminate */
            ERROR("unterminated dst");
      }  else  {
            ERROR("truncation");
      }
```

ISO/IEC 24731: string and memory functions

- `strcpy_s` `strncpy_s` `memcpy_s` `strcat_s` `strncat_s` `memmove_s`
- Like standard counterpart, except all include an additional parameter for the length of the destination buffer
- Run-time constraint failure if destination
- If error
  - Null-terminates destination buffer, null fills buffer for mem functions

ISO/IEC 24731: string and memory functions

- Very easy to convert typical unsafe code
  - Add _s to function name
  - Add destination buffer size parameter

Proper usage: concat s1, s2 into dst

```
/* program will abort on error */
strcpy_s(dst, dstSize, s1);
strcat_s(dst, dstSize, s2);
```

ISO/IEC 24731: `printf_s` family

- `%n` conversion not allowed as it is often used by attackers to write to arbitrary memory
- Null arguments to `%s` are a violation
- Use `sprintf_s` instead of `snprintf_s`
- `snprintf_s` truncates just like `snprintf`

Proper usage: concat s1, s2 into dst
`/* program will abort on error */`
`sprintf_s(dst, dstSize, "%s%s", s1, s2);`

THE UNIVERSITY of WISCONSIN MADISON

SDSC

ISO/IEC 24731: `scanf_s` family

- `%s`, `%c` and `%[...]` now require the argument to be a pointer to a buffer followed by the size of the buffer
- Null arguments are a violation
- Still requires synchronizing result, format string and arguments

Preventing Buffer Overflows in C++

- **Don't use pointers, C-style string, or C-arrays**
- `std::string`
  - Buffer, length and current size are encapsulated together
  - Dynamically sized
  - Provides many useful and efficient methods
- `std::vector<>`
  - Dynamically sized array
  - Safe except `operator[]` (use `at` method for safety)
- `std::array<>` (new in C++ TR1)
  - Fixed size array

Proper usage: concat s1, s2 into dst
`dst = s1 + s2;`

Potential Problems with C++ Strings

- System call and libraries expect C-strings
  - `c_str` method will return a constant C-string pointer
  - Valid only until string is modified
  - Nulls are allowed
    - When converted to C-string everything after the null is essentially lost
    - If tests are done on C++-string and used as a C-string these may be different
    - Same problem in other languages such as Perl
- Denial of service if length not restricted

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
  - Not performing data validation on user input
  - Not properly quoting user data to prevent misinterpretation of metacharacters when they can't be rejected during validation

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)
  – Uses SQL commands created at run-time
- SQL fragments must not be accepted from user; create parsable language and translate to SQL if needed

SQL Injection Attacks

- Dynamically generated SQL without validation or quoting is vulnerable

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

- Quoting values is safe if done correctly

```
$u = " \\'; drop table t --";     # perl eats one \
$u =~ s/'/''/g;                   # quote (' -> '')
$sth = $dbh->do("select * from t where u = '$u'")
--  select * from t where u = ' \''; drop table t --'
```

- Previous example is correct in standard SQL, but incorrect in systems that allow \-escapes

SQL Injection Mitigations

- Use prepared statements (no quoting)

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

- Use library functions to perform quoting

```
$sth = $dbh->do("select * from t where u = "
                 . $dbh->quote($u))
```

- Views can be used to limit access to data
- Stored procedures can help, but not if they dynamically create and execute SQL
- Restrict rights of database account to minimum required

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`
- Usually done to start another program
  - That has no C API
  - Out of laziness

Command Injection Mitigations

- Check user input for metacharacters
- Quote those that can't be eliminated or rejected
  - replace single quotes with the four characters, `'\''`, and enclose each argument in single quotes
- Beware of program argument injections, allowing arguments to begin with `"-"` can be dangerous
- Use `fork`, drop privileges and `exec` for more control
- Avoid if at all possible
- Use C API if possible

Perl Command Injection Danger Signs

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

Perl Command Injection Danger Signs

- Vulnerable to shell interpretation

```
open(C, "$cmd|")          open(C, "-|", $cmd)
open(C, "|$cmd")          open(C, "|-", $cmd)
`$cmd`                    qx/$cmd/
system($cmd)
```

- Safe from shell interpretation

```
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 = $ot)   =~ s/'/'\\''/g;
open(CMD, "|/bin/mail -s '$qSub' '$qTo'");
```
  - Safe from command injection
- `open(cmd, "|-", "/bin/mail", "-s", $sub, $to);`
  - Also safe and simpler

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**

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

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

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

Format String Injections

- User supplied allowed 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`

Cross Site Scripting (XSS)

- Attacker supplied data passed to through a web server to be delivered to a victim
  - Can be part of the URL
  - Stored by attacker from previous interaction with web server
- Injected javascript in HTML can be used to modify HTML interpreted by user's browser
- Allows stealing of cookies and redirecting page
- Web server needs to escape all user supplied data

Other Injections

- Line delimited log and data files
  - Need to verify user supplied data does not contain the delimiter character
  - Escape or reject if it does
  - If not, user can inject records in the file
- User supplied data to XPath queries
- User supplied data used in LDAP queries

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'`

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
- `size_t` is unsigned, `ptrdiff_t` is signed
- Using an integer type instead of the correct integral typedef type
- Not assigning values to a variable of the correct type before data validation, so the validated value is not the same as the value used

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

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
  - Using threads without proper synchronization including non-thread (non-reentrant) safe functions
  - Performing non-atomic sequences of operations on shared resources (file system, shared memory) and assuming they are atomic
  - Signal handlers

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
- To be safe path should only be used once to create a file descriptor (fd) which is a handle to internal identifier
- Other checks should be done on fd

Race Conditions Checking File Properties

- Use the path to check properties of a file, and then open the file (also called time of check, time of use TOCTOU)
  - `access` followed by `open`
    - Safe to just set the effective ids and then just open the file
  - `stat` followed by `open`
    - Safe to `open` the file and then `fstat` the file descriptor

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

Race Condition Creating a File

- Want to atomically check if file exists and create if not, or fail if it exists
- Common solution is to check if file exists with `stat`, then `open` if it doesn't
- Open a file or create it if does not exist
  - `creat(fname, mode)`
    `open(fname, O_CREAT|O_WRONLY|O_TRUNC, mode)`
- Must use `O_CREATE|O_EXCL` to get desired property
- Never use `O_CREATE` without `O_EXCL`

Race Condition Creating a File

- `open` also fails if the last component of the path is a symbolic link when using `O_CREATE|O_EXCL`
- `fopen` **never uses** `O_EXCL`
  - Only use for read mode
  - For append or write modes use `open` and `fdopen` to create a `FILE*` from a file descriptor
- C++ iostreams never use `O_EXCL`
  - No standard way to get iostream from fd
  - Use use non-standard extension
  - Use library that can create a stream from a fd, such as http://www.boost.org/libs/iostream

Race Condition Creating a File

- If you want to open or create like `O_CREAT` without `O_EXCL` use the following:

```
f = open(fname, O_CREAT|O_EXCL|O_RDWR, mode);
if (f == -1 && errno == EEXIST)  {
    f = open(fname, O_RDWR)
}
```

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);
```

THE UNIVERSITY of WISCONSIN MADISON

SDSC

Race Condition Temporary Files

- Temporary directory (/tmp) is the bad part of town for the file system
- Any process can create a file there
- Usually has the sticky bit set, so only the owner can delete their files
- Ok to create true temporary files here
  - Created, immediately deleted, and only accessed through the original file descriptor
  - Storage vanishes when file descriptor is closed
- If you must use the /tmp directory at least create a secure bunker by creating a restricted directory to store your files

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** / time / **Attackers Action**

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

Safe Version

```
fd = mkstemp(s)
f = fdopen(s, "w+")
```

Race Condition Examples

time

Your Actions

```
s="/bin/rooty"
// umask is 0000
F = fopen(s, "w+")
// mode is 0666 here
chmod(s, 06555)
// setuid plus rx

// owner is root
chown(s, uid, gid)
```

Attackers Action

```
s="/bin/rooty"
f = open(s, O_WRONLY)
write(f, evilExec, sz)
close(f);
execl(f, 0)
// evilExec is now run
```

Safe Version

```
fd = fopen(s, "w", 0)
fchown(fd, uid, gid)
fchmod(fd, 06555)
F = fdopen(s, "w+")
```

THE UNIVERSITY of WISCONSIN MADISON

SDSC

ISO/IEC 24731: `fopen_s` family

- Permissions of created files
  - `Only allow access to owner`
- `"u…"` modes
  - Behaves like fopen in that permissions of a newly created file are only affected by the umask
  - Must go before all other mode characters
- Null arguments are a violation
- Doesn't fix lack of `O_EXCL` when creating, so should still use `open` and `fdopen`

ISO/IEC 24731: temporary files

- `tmpfile_s(FILE** f)`
  - Creates a file
  - Permissions only allows access to owner
  - File deleted by time of exit
- `tmpnam_s`
  - Create non-existing temporary file names
  - Buffer includes length
  - Use of name can lead to race condition unless precautions are taken

ISO/IEC 24731: Reentrant safe functions

- `strtok_s`
  - Like `strtok_r` plus size of buffer
- `getenv_s`
  - Copy value to buffer with size
- `asctime_s ctime_s`
  - Copy value to buffer with size
- `gmtime_s localtime_s`
  - Like `_r` versions with non-null constraint
- `strerror_s strerrorlen_s`
  - Copy value to buffer with size
  - `strerror_s` truncates if not big enough

Other Dangers in the File System

- Some file systems are case-insensitive, but might be case-preserving
- Any user can create a hard link to any file, even if permissions don't allow any access
- Some file systems have files with multiple forks
  - Special path or API to get at non-default fork
  - MacOS: f/..namedfork/data ≡ f ≡ f/.__Fork/data
  - Windows: f ≡ f$DATA

Other Dangers in the File System

- Data can be hidden in other forks if only using standard API
- Some file systems support extended file attributes that are key, value pairs that can used to hide data
- Other privilege systems may be in use that change the privileges a user would appear to have from the standard POSIX model
  - AFS
  - Extended ACLs

Non-canonical Forms

- If one value can be encoded in multiple different forms they must be converted to a unique canonical form before comparison
  - Paths: use (device, inode) pair, or convert to a canonical form using realpath
  - Usernames and groups: use uid and gid
  - utf: convert to utf-32 or canonical form
  - HTML & URL encoded: decode
  - Case insensitive: convert to all lower (some languages lose info if converted to upper)

Non-canonical Forms

- In weakly typed language, such as a shell or Perl, where a value can be a number or string use the correct comparison operator
  - Comparing numbers lexically is bad
    - `"100" le "2"`
    - `"000" ne "0"`
  - Comparing strings numerically is bad
    - `"111111" > "9sdflkj"`
    - `"000" == "0abc"`
    - `"xyz" == "abc"`

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
- Everything executable requires must be 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 user the owner of the file is equivalent to the privileged user
  - Owned by privileged user
  - 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 except from trusted users

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

- **Description**
  - A program’s environment is a list of strings that a program is allowed to interpret. Libraries are also allowed to use the environment to alter their behavior. Since changes to the environment can alter the execution of a program, library code, and spawned programs, the environment must be carefully controlled.
- **General causes**
  - Not sanitizing the environment
  - Allowing user input to alter the environment
  - Not fully specified as to what happens when multiple values with the same name, or value without '=' in it

Environment Mitigation

- Record needed values of the environment, sanitize them, clear the environment, add only necessary values, discard others
- Don't make assumptions about size of values
- Don't allow code from the user to set environment
- Use `execle` or `execve` to start a process with user supplied environment variables
- Use `setenv` instead of `putenv`

Environment Mitigation

- `PATH`: list of directories to search for executables given as just a name
  - Only used by `execlp` and `execvp`
  - Use `execle` or `execve` and full paths
  - Set `PATH` to something safe `/bin:/usr/bin`
- `LD_LIBRARY_PATH`: list of directories to search for shared libraries, could be used to inject a library into your process

Denial of Service

- Description
  - Programs becoming unresponsive due to over consumption of a limited resource or unexpected termination.
- General causes
  - Not releasing resources
  - Crash causing bugs
  - Infinite loops or data causing algorithmic complexity to consume excessive resources
  - Failure to limit data sizes
  - Failure to limit wait times
  - Leaks of scarce resources (memory, file descriptors)

Information Leaks

- Description
  - Inadvertent divulgence of sensitive information
- General causes
  - Reusing buffers without completely erasing
  - Providing extraneous information that an adversary may not be able to otherwise obtain
    - Generally occurs in error messages
    - Give as few details as possible
    - Log full details to a database and return id to user, so admin can look up details if needed

Information Leaks

- General causes (cont.)
  - Timing attacks where the duration of the operation depends on secret information
  - Lack of encryption when using observable channels
  - Allowing secrets on devices where they can't be erased such as swap space (use `mlock`) or backups

Memory Allocation Errors

- Description
  - For languages with explicit dynamic memory allocation (C and C++), if the internal heap data structures can be corrupted, they can lead to arbitrary execution of code.
- General causes
  - Buffer overflows
  - Releasing memory multiple times
  - Releasing invalid pointers for the allocator (from a different allocator or garbage pointers)

Memory Allocation Mitigations

- In C++ use the STL, `auto_ptr`, and destructors to let the compiler free memory
- Match `malloc/free`, `new/delete`, and `new[]/delete[]`
- Allocate and deallocate memory for an object "close" to each other
- Use tools such as Purify or valgrind to find problems

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 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
- 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

Let the Perl Compiler Help

- `-w` - produce warning about suspect code and runtime events
- `use strict` - fail if compile time
- `use Fatal` - cause built-in function to raise an exception on error instead of returning an error code
- `use diagnostics` - better diagnostic messages

Perl Taint Mode

- Taint mode (-T) prevents data from untrusted sources from being used in dangerous ways
- Untrusted sources
  - Data read from a file descriptor
  - Command line arguments
  - Environment
  - User controlled fields in password file
  - Directory entries
  - Link referents
  - Shared memory
  - Network messages
- Environment sanitizing required for `exec`
  - `IFS PATH CDPATH ENV BASH_ENV`

Resources

- Viega, J. & McGraw, G. (2002). *Building secure software: How to avoid security problems the right way*. Addison-Wesley.
- Seacord, R. C. (2005). *Secure coding in C and C++.* Addison-Wesley.
- McGraw, G. (2006). *Software security: Building security in*. Addison-Wesley.
- Dowd, M., McDonald, J., & Schuh, J. (2006). *The art of software assessment: Identifying and preventing software vulnerabilities*. Addison-Wesley.

Questions