CSC 482/582: Computer Security

Input Validation
Topics

1. Attack Surface
2. Validation Methodology
3. Entry Points
4. Validation Strategies
5. Secure Programming References
The **attack surface** of a system consists of the ways in which a **threat** can **enter** the system.

**Methods** are code components that receive input. **Channels** are avenues of communication (sockets, environment, keyboard, files, etc.) **Data** are the actual input strings.
Defending Attack Surface

- Minimize the attack surface
  - Design application so that the smallest number of required entry points exist.
  - Limit access to required entry points to designated networks (firewall) and personnel (authentication).
  - Design application so that entry points accept well-defined input types that are not more complex languages than needed.

- Verify attack surface
  - Verify that no unexpected entry points exist.
  - Verify that all entry points that should require authentication do require it.

- Securely validate input at all entry points.
Input Validation Principles

Never trust input.
- Assume dangerous until proven safe.

Prefer rejecting data to filtering data.
- Difficult to filter out all dangerous input

Every component should validate data.
- Trust is transitive.
- Don’t trust calling component.
- Don’t trust called component: shell, SQL
Trust Boundaries

- Environment variable
- Chokepoint
- Trust boundary
- Service
  - Trust each other as they are both inside the trust boundary
  - Config data
- Service Data
- Chokepoint

CSC 482/582: Computer Security
Validation Techniques

Indirect Selection
- Allow user to supply index into a list of legitimate values.
- Application never directly uses user input.
- Most secure but cannot be applied to all inputs.

Whitelist
- List of valid patterns or strings.
- Input rejected unless it matches list.
- Secure and can be applied to all inputs.

Blacklist
- List of invalid patterns or strings.
- Input reject if it matches list.
- Least secure.
Validation Actions

Sanitize
- Attempt to fix input by removing dangerous parts.

Reject
- Refuse to use invalid input.

Reject with Explanation
- Explain problems with input to user.
- Refuse to use invalid input.

Log
- Record invalid input in log file.

Alert
- Send an alert to an administrator about input.
Check Input Length

Long input can result in buffer overflows.

- Can also cause DoS due to low memory.

Truncation vulnerabilities

- 8-character long username column in DB.
- User tries to enter ‘admin x’ as username.
- DB returns no match since name is 9 chars.
- App inserts data into DB, which truncates.
- Later SQL queries will return both names, since MySQL ignores trailing spaces on string comparisons.
Check Input Type

Is input the type you expect?

- Integer
- Temperature (integer within specified range)
- Money value (fixed point)
- Name (alphabetic string allowing ‘, -, spaces)

Compiler type checking doesn’t solve this for you:

- Simple input types (integers and the like) are checked.
- But strings that represent complex types like HTML, XML, JSON, SQL, etc. are not.
Is input in correct format?

- Phone number (xxx-xxx-xxxx format)
- International phone number (more options)
- Credit card number (format, Luhn checksum)
- URL
- E-mail address

Libraries may do this, but check that they actually work:

- Many web frameworks have validator classes.
- Parsers for XML, JSON, etc. can do checking themselves.
Minimal Computational Power

Input languages should be at the lowest level of the Chomsky hierarchy possible.

- The higher the language level, the more attackers can do.

Input validation techniques must be capable of handling the full input language.

- Example: Regular expressions cannot validate a context-free input language, which includes any language that supports nested matching braces.
### Regular Expressions

#### Anchors
- `^` Start of line
- `\A` Start of string
- `$` End of line
- `\Z` End of string
- `\b` Word boundary
- `\B` Not word boundary
- `<` Start of word
- `>` End of word

#### Sample Patterns
- `([A-Za-z0-9-]+)` Letters, numbers and hyphens
- `(\d{1,2})\d{1,2}\d{4}` Date (e.g. 21/3/2006)
- `([^\s]+@\.(jpg|gif|png))\.\d` jpg, gif or png image
- `([^1-9]{1}$[^1-4]{1}[0-9]{1}$[^0-9]{1}$[^50$)` Any number from 1 to 50 inclusive
- `#([A-Za-z0-9-])3([A-Za-z0-9-])3` Valid hexadecimal colour code
- `([=\-\s]+d)([=\-\s][a-z])([=\-\s][A-Z]).(8,15)` 8 to 15 character string with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
- `(\w+@[a-zA-Z]+\.|[a-zA-Z]{2,6})` Email addresses
- `(\<\/?\{[^\}]\*>\>)` HTML Tags

#### Character Classes
- `\c` Control character
- `\s` White space
- `\S` Not white space
- `\d` Digit
- `\D` Not digit
- `\w` Word
- `\W` Not word
- `\xhh` Hexadecimal character hh
- `\Oxxx` Octal character xxx

#### POSIX Character Classes
- `[:upper:]` Upper case letters
- `[:lower:]` Lower case letters
- `[:alpha:]` All letters

#### Quantifiers
- `*` 0 or more
- `??` 0 or more, ungready
- `+` 1 or more
- `??` 1 or more, ungready

#### Ranges
- `.` Any character except new line (`\n`) +
- `(a|b)` a or b +
- `(...)` Group +
- `(?::...)` Passive Group +
- `[abc]` Range (a or b or c) +
- `[^abc]` Not a or b or c +
- `[a-q]` Letter between a and q +
- `[A-Q]` Upper case letter +
- `between A and Q` +

---

**Note:** These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.
Regular Expression Variants

Common Types

- **POSIX BRE**: POSIX standard basic regular expressions. Can use with grep -b.
- **POSIX ERE**: POSIX extended regular expressions add +, ?, |. Can use with grep -E.
- **PCRE**: Perl-Compatible regular expressions, supported by many languages and tools, can recognize languages beyond regular expressions and therefore can match braces. etc. Use with grep -P.

Character Types

- Some regular expression libraries support Unicode and/or multiple character sets; other do not.
Input Validation with REs

1. Write regular expression describing valid input.
   - Be strict. Avoid accepting input that is not valid.
   - Use anchors: ^ at the beginning and $ at the end, to avoid mistakes like attempting to match file suffixes with /.zip/, which accepts bzip2.exe, instead of /\.*.zip$/

2. Test if input matches regular expression.
   - Reject input if it does not match.
   - Process input if it does match.
Entry Points

1. Command line arguments
2. Environment variables
3. File descriptors
4. Signal handlers
5. Strings
6. Paths
7. Shell input
8. Web application input
9. Database input
10. Other input types
Command Line Arguments

Available to program as **argv.
`execve()` allows user to specify arguments.
May be of any length
- even program name, `argv[0]`
- `argv[0]` may even be NULL
Environment Variables

Default: inherit parent’s environment.

`execve()` allows you to specify environment variables for exec’d process.
- environment variables can be of any length.

Telnet environment propagation to server
- Server receives client shell’s environment.
- Server runs setuid program `login`.
- `ssh` may use user’s `~/.ssh/environment` file.
Dangerous Environment Variables

**LD_PRELOAD**
- Programs loads functions from library specified in LD_PRELOAD before searching for system libraries.
- Can replace any library function.
- Setuid root programs don’t honor this variable.

**LD_LIBRARY_PATH**
- Specify list of paths to search for shared libs.
- Store hacked version of library in first directory.
- Modern libc implementation disallow for setuid/setgid.
PATH

- Search path for binaries
- Attacker puts directory with hacked binary first in PATH so his `ls` used instead of system `ls`
- Avoid “.” as attacker may place hacked binaries in directory program sets CWD to

IFS

- Internal field separator for shell
- Used to separate command line into arguments
- Attacker sets to “/”: `/bin/ls` becomes “`bin`” and “`ls`”
Environment Storage Format

Access Functions

- `setenv()`, `getenv()`

Internal Storage Format

- array of character pointers, NULL terminated
- string format: “NAME=value”, NULL term
- Multiple environment variables can have same name.
- Did you check the same variable that you fetched? First or last variable that matches?
Securing Your Environment

/* BSS, pp. 318-319 */
extern char **environ;
static char *def_env[] = {
    "PATH=/bin:/usr/bin",
    "IFS= \t\n",
    0
};
static void clean_environment() {
    int i = -1;
    while( environ[++i] != 0 );
    while(i--)
        environ[i] = 0;
    while(def_env[i])
        putenv(def_env[i++]);
}
Securing Your Environment

Secure Environment in Shell

```
/usr/bin/env – PATH=/bin:/usr/bin IFS=“ \t\n” cmd
```

Secure Environment in Perl

```
%ENV = (
  PATH => “/bin:/usr/bin”,
  IFS => “ \t\n”
);
```
File Descriptors

Default: inherited from parent process
- `stdin`, `stdout`, `stderr` usually fd’s 0, 1, and 2

Parent process may have closed or redirected standard file descriptors

Parent may have left some fd’s open
- Cannot assume first file opened will have fd 3

Parent process may not have left enough file descriptors for your program
- Check using code from BSS, p. 315
Signal Handlers

Default: inherited from parent process.

/* BSS, p. 316 */
#include <signal.h>

int main( int argc, char **argv ) {
    int i;
    for(i=0; i<NSIG; i++)
        signal(I, SIG_DFL);
}
String Inputs

Aspects of string input

- Length
- Character encoding

Encodings describe how bits map to chars

- ASCII 7-bit encoding for English.
- ISO-8859-1 (Latin 1) 8-bit encoding
  - Compatible with ASCII
  - New chars for other Latin alphabet languages
  - Windows-1252 variant common
- Unicode family of encodings for all languages
Universal Character Set (UCS)

Can represent all chars for all languages.
  - Represents characters as code points.
  - Planes are groups of 65,536 numerical values that represent code points.
  - 1,112,064 code points from 17 planes are accessible with current encodings.

Basic Multilingual Plane (BMP)
  - The first 65,536 UCS characters.
  - UCS-2 was an early 16-bit encoding to represent only characters from the BMP.

Supplementary Ideographic Plane
  - Contains many CJK ideographs.
Map of the BMP
Homograph Attacks

Spoofing attack that relies on fact that different characters are identical visually.

- In ASCII, O and 0, 1 and l are identical in some fonts.
- See [http://www.unicode.org/Public/security/revision-05/confusables.txt](http://www.unicode.org/Public/security/revision-05/confusables.txt) for a list.

Example: Cyrillic has 11 homographs with Latin

- U+0430 is ‘a’ in Cyrillic alphabet
- U+0061 is ‘a’ in Latin alphabet
- Allows attacker to spoof paypal.com.

International Domain Names enabled in 2003.

- IDNs stored in DNS using Punycode ASCII.
Unicode Encodings

UTF-8
- Variable length 8-, 16-, 24-, or 32-bit encoding
- Can represent any char on the 17 plans.
- Backwards compatible: first 128 chars are ASCII.
- Over half of web pages use UTF-8 encoding.

UTF-16
- Variable length 16-bit or 32-bit encoding
- Can represent any char on the 17 planes.
- Used in Windows API since W2k.
- Java added UTF-16 support in Java 5.
- Special syntax for non-BMP in most languages.
## UTF-8

<table>
<thead>
<tr>
<th>Bits</th>
<th>Last code point</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>U+007F</td>
<td>0xxxxxx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>U+07FF</td>
<td>110xxxx</td>
<td>10xxxxx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>U+FFFF</td>
<td>1110xxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>U+1FFFFF</td>
<td>11110xxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>U+3FFFFFFF</td>
<td>111110xx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>U+7FFFFFFFFF</td>
<td>1111110x</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
<td>10xxxxx</td>
</tr>
</tbody>
</table>

- Not all bit sequences valid, esp. overly long sequences that can represent same character using techniques in each of 6 rows above to bypass input validation.
- Check for validity of UTF-8 strings before checking if strings match whitelist.
Paths

If attacker controls paths used by program
- Can read files accessible by program.
- Can write files accessible by program.

Vuln if access is different than attackers
- Privileged (SETUID) local programs.
- Remote server applications, including web.

Directory traversal
- Use “..../..../..” to climb out of application’s directory and access files.
Path Traversal Example

```php
<?php
$template = 'red.php';
if (isset($_COOKIE['TEMPLATE']))
    $template = $_COOKIE['TEMPLATE'];
include ('/home/users/phpguru/templates/' . $template);
?>
```

GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../../../../etc/passwd

HTTP/1.0 200 OK
Content-Type: text/html
Server: Apache

root:fi3sED95ibqR6:0:1:System Operator:/bin/ksh
daemon:*:1:1::/tmp:
phpguru:f8fk3j1OiF31.:182:100:Developer:/home/users/phpguru/
The Problem

How to make correct access control decisions when there are many names for a resource?

- config
- ./config
- /etc/program/config
- ../program/config
- /tmp/./etc/program/config
Canonicalization

- Canonical Name: standard form of a name
  - Generally simplest form.
  - Canonicalize name then apply access control

- UTF-8 canonicalization in Java
  - String s = "\uFE64" + "script" + "\uFE65";
  - s = Normalizer.normalize(s, Form.NFKC);

- APIs to canonicalize pathnames
  - C: realpath()
  - Java: getCanonicalPath()
Common Naming Issues

- . represents current directory
- .. represents previous directory
- Case sensitivity
- Windows allows both / and \ in URLs.
- Windows 8.3 representation of long names
  - Two names for each file for backwards compat.
- Trailing dot in DNS names
  - www.nku.edu. == www.nku.edu
- URL encoding
Path Traversal Encodings

URL Encodings
- \%2e\%2e\%2f translates to ../
- \%2e\%2e\/ translates to ../
- ..\%2f translates to ../
- \%2e\%2e\%5c translates to ..\\

Unicode Encodings
- \%c1\%1c translates to /
- \%c0\%af translates to \

Win/Apache Directory Traversal

Found in Apache 2.0.39 and earlier.

To view the file `winnt\win.ini`, use:

```
http://127.0.0.1/error/%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%5c%2e%5c%2e%5c\cwinnt%5c\cwin.ini
```

which is the URL-encoded form of

```
http://127.0.0.1/error/\..\..\..\..\..\winnt\win.ini
```
Command Injection

Find program that invokes a subshell command with user input

UNIX C: `system()`, `popen()`, ...
Windows C: `CreateProcess()`, `ShellExecute()`
Java: `java.lang.Runtime.exec()`
Perl: `system()`, ````, `open()`

Use shell meta-characters to insert user-defined code into the command.
UNIX Shell Metacharacters

`command` will execute command
`;` separates commands
`|` creates a pipe between two commands
`&&` and `||` logical operators which may execute following command
`!` logical negation—reverses truth value of test
`-` could convert filename into an argument
`*` and `?` glob, matching files, which may be interpreted as args: what if “-rf” is file?
`#` comments to end of line
/* Mail to root with user-defined subject */
int main( int argc, char **argv ) {
    char buf[1024];
sprintf( buf, "/bin/mail -s %s root </tmp/message", argv[1] );
system( buf );
}
Command Injection in C

How to exploit?

./mailprog `path/to/hacked_bin`
/path/to/hacked_bin will be run by mailprog

How to fix?

Verify input matches list of safe strings.
Run /bin/mail using fork/exec w/o a subshell.
Command Injection in Java

```java
String btype = request.getParameter("backuptype");
String cmd = new String("cmd.exe /K \"c:\\util\\rmanDB.bat \n"+btype+"&&c:\\utl\\cleanup.bat\"\")
System.Runtime.getRuntime().exec(cmd);
```
Command Injection in Java

How to exploit?
- Edit HTTP parameter via web browser.
- Set `bype to be "&& del c:\\dbms\\*.*"

How to defend?
- Verify input matches list of safe strings.
- Run commands separately w/o cmd.exe.
Database Input

SQL Injection

- Most common flaw in database input parsing.
- Don’t pass unvalidated data to database.
- Whitelist for known safe character set.
  - Alphanumerics
  - How many symbols do you need to accept?

Don’t trust input from database.

- Check that you receive expected # of rows.
- Check for safe data to avoid stored XSS and second order SQL injection attacks.
Other Inputs

Default file permissions

- `umask(o66);`

Resource Limits

- May suffer DoS if parent imposes strict limits on CPU time, # processes, file size, stack size.
- Use `setrlimit()` to limit core dump size to zero if program ever contains confidential data in memory, e.g., unencrypted passwords.
Separate Usability and Security Validation

Usability Validation helps legitimate users
- Catch common errors.
- Provide easy to understand feedback.
- Client-side feedback is helpful for speed.

Security Validation mitigates vulnerabilities
- Catches potential attacks, including unusual, unfriendly types of input.
- Provide little to no feedback on reasons for blocking input.
- Must be done on server. Cannot trust client.
Choke Points

- Require all input go through a single choke point
  - Ignore or reject other input sources.
  - Validate all input at choke point.
  - Example: Send all URLs to a single server program.

Problems

- Some systems require multiple input sources. If so, reduce to as few sources as possible.
- Some validation decisions need precise context to be made correctly, especially semantic validation, so choke point may be limited to length, type, and syntax validation.
Wrap Dangerous Functions

Input is context sensitive.
- Need more context than is available at front end.

Solution: create secure API
- Apply context-sensitive input validation to all input.
- Maintain input validation login in one place.
- Ensure validation always applied.
- Use static analysis to check for use of dangerous functions replaced by API.
Key Points

1. Validate input from all entry points. CLI args, env vars, config files, database, etc.
2. Use the strongest possible technique.
   1. Indirect Selection
   2. Whitelist
   3. Blacklist
3. Reject bad input, don’t attempt to fix it.
4. Validate length, type, syntax, and semantics.
5. Input validation strategies
References