Secure Programming Techniques

Input validation

- Input checking
- Resource usage
- Most attacked applications
- What to validate
- What makes up input?
- Special input in web applications
- Example: Perl and taint mode
- Taint propagation in general
- Transforming input
- Example: getting input right
- Example: readlink
Secure Programming Techniques

Input checking

- Program must cope with any input given to it, including badly formed input
- Input must conform to allowed pattern
- Input must conform to allowed size constraints
- Input is more than directly read data — also anything else from the system
- Check the usage of CPU time, RAM, I/O, network usage (danger of DoS!)
- Also test for false acceptance and false filtering, not only for acceptance of good input
- Validate input at each module border, not only in user interface
- Allowed patterns versus forbidden patterns?
Resource usage

- Correct allocation of resources
- Correct usage of resources
- Correct freeing of resources
- Watch your resource usage
  - limits (ulimit, quota, …)
    - stop execution when exceeded
- Don’t take assumptions about input for granted here
- Good predictability of resource usage might be a problem by itself
Most attacked applications

- Viewers of data coming from untrusted sources (browser, editor)
  - Executing any input as a program is dangerous!
  - Macros — a special kind of programs
- Programs running with administrative privileges
- Local services
- Network services
- Web applications!!!
- Applets
- `setuid/setgid` programs in Unix
What to validate

• Validate all input
  – Make it easy to verify that all input is validated before it is used
  – Strong typing
  – Length checks
  – Range checks
  – Syntax (e.g. email address)

• Validate input from all sources

• Establish trust boundaries
  – Store trusted and untrusted data separately
  * Sessions!
What makes up input?

- User-entered data
- Program arguments (command line)
  - Garbage as $argv[0]$ (program name)?
- Environment variables
  - LD_PRELOAD, LD_LIBRARY_PATH; IFS; PATH, CDPATH; ENV, BASH_ENV, . . .
- Configuration files, system properties
  - Apache .htaccess and RewriteRule buffer overflow
- Registry values
- Database queries (incl. number of rows in result)
  - Is $100$ good for book price?
  - What about 'a string instead' or 1.0E35?
What makes up input?

- Network services (DNS, ...)
- File handles
  - Having standard input, standard output, standard error always open might be a bad assumption
- File contents, including temporary files
  - Even the ones you should have just created yourself
- Signals
  - Unexpected signals can break the program
- umask
  - Set strict permission on file create if needed
- Current working directory
  - Might be anywhere — use absolute paths if needed
Special input in web applications

- **URL**

- **Query body**
  - Included form contents supposedly encoded by browser
  - Hidden form variables

- **URL encoding**
  - Standard hex encoding %20
  - Different UTF-8 bytestreams decoding to the same value

- **Any HTTP headers**
  - Cookies, Referer

- **Client side validation is insufficient**
  - Don’t mix up usability and security!
Example: Perl and taint mode

- **Taint mode** — environment is automatically considered insecure
  - Used automatically for setuid and setgid programs
  - Can be switched on by hand with `-T`
  - Often used for CGI programs

- You must not use data derived from outside your program to affect something else outside your program
  - **Exceptions:** `print`, `syswrite`, `symbolic names`

- Values derived from tainted data are also tainted
- Tainted data must be explicitly validated
Perl: removing taint

- The only way to get rid of taintedness is to match regular expressions and extract subexpressions from the result:

```perl
if ($input =~ /^([0-9]+@[0-9A-Za-z.]+)$/) {
    $input = $1;
} else {
    die "Invalid input: $input";
}
```

- By the way, what’s wrong with the example code?
Perl and tainted data

- Taintedness is remembered for each scalar value
- Hash keys are never tainted
- External data that is considered to be tainted:
  - Command line
  - Environment variables
  - Locale settings (charset name etc)
  - User information (gecos fields)
  - Information from shared memory and message queues
  - Data read from files and sockets
  - Filesystem metadata (directory tree etc)
- In addition, writability of directories in PATH and @INC are also checked in taint mode
Taint propagation in general

- Dataflow analysis:
  - Source rules — where tainted data enters the system
  - Entry rules — similar but invoked by the attacker
  - Pass-through rules — ways of manipulating tainted data
  - Cleanse rules — ways for removing taint
  - Sink rules — program locations that should not receive tainted data

- Static or dynamic taint tracking

- Binary flag or different taint flags
Taint rule example

- **Source rule:**
  
  Function:
  
  `javax.servlet.http.HttpServletRequest.getParameter()`
  
  Postcondition:
  
  return value is tainted

- **Sink rule:**
  
  Function:
  
  `javax.servlet.http.HttpSession.setAttribute()`
  
  Precondition:
  
  arguments must not be tainted
Transforming input

- Beware of too eager transformation of input after validating it
  - At each module boundary?

- Family of double decode vulnerabilities
  - Data is decoded again after successful validation and filtering
Double decoding vulnerability

- Example: MUST NOT give access to *.inc, attacker wants config.inc

- Simple filter rule: `deny /\.*inc$/`
  - What about config.INC if our file system is case-insensitive?

- Better filter rule: `deny /\.*inc$/i/
  - Input "config.INC" filtered out, good
  - Input "config%2Einc" is URL-encoded, passes
  - Later URL-decoded to be sure \(\Rightarrow\) badness
  - Filter out % or change code to decode only in one place?

- What about "config.inc%00" (with terminating zero byte)?
Example: getting input right

- Let’s input a text string in plain C:

```c
char input[80];
gets(input);
```
Example: getting input right

• Try 2:

```c
char input[80];
fgets(input, 80, stdin);
```
Example: getting input right

- What about C++?

```cpp
char input[80];
cin >> input;
```
**Example: getting input right**

- Try 2 — automatic allocation (what’s wrong here?)

```cpp
string input;
cin >> input;
```
Example: getting input right

- Try 3 — with width limit:

```cpp
string input;
cin >> setw(80) >> input;
```
Example: input size

- Java is free of buffer overflows but memory consumption is a problem there too:

```java
String input;
input = bufferedReader.readLine();
```
Input validation with regular expressions

- Regular expressions are a convenient way for input validation
  - E-mail address regexp, anyone?
- Regular expression are also good for input size limiting
Example: e-mail address

- RFC 2822: double quotes for usernames, [ ] for mail routing
  - Let’s ignore them for now

- `^[A-Z0-9\.\_%+-]+@[A-Z0-9.-]+\.\.[A-Z]+$`
- `^[A-Z0-9\.\_%+-]+@[A-Z0-9.-]+\.\.[A-Z]{2,6}$`
- `^[A-Z0-9\.\_%+-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$`
- `^[A-Z0-9\.\_%+-]{1-32}@(?:[A-Z0-9-]{1,20}\.){1,10}[A-Z]{2,6}$`
Example: readlink

- Try 1: string termination error — readlink() does not put a null terminator at the end of buf

```c
void badReadlink1(char* path) {
    char buf[PATH_MAX];
    int ret = readlink(path, buf, PATH_MAX);
    printf("file is: %s\n", buf);
}
```
Example: readlink

- Try 2: missing error handling + buffer overflow — readlink() may return an error code

```c
void badReadlink2(char* path) {
    char buf[PATH_MAX];
    int ret = readlink(path, buf, PATH_MAX);
    buf[ret] = 0;
    printf("file is: %s\n", buf);
}
```
Example: readlink

- Try 3: off-by-one buffer overflow — readlink() may return PATH_MAX, causing the null terminator to be written one byte past the end of the array

```c
void badReadlink3(char* path) {
    char buf[PATH_MAX];
    int ret = readlink(path, buf, PATH_MAX);
    if (ret <= 0) {
        perror("error reading link");
    } else {
        buf[ret] = 0;
        printf("file is: %s\n", buf);
    }
}
```
Example: readlink

- Try 4: logic error — file names may be longer than PATH_MAX, so there is no guarantee that buf contains the complete path name

```c
void badReadlink4(char* path) {
    char buf[PATH_MAX];
    int ret = readlink(path, buf, PATH_MAX-1);
    if (ret <= 0) {
        perror("error reading link");
    } else {
        buf[ret] = 0;
        printf("file is: %s\n", buf);
    }
}
```