SWE 681 / ISA 681
Secure Software Design & Programming:
Lecture 2: Input Validation

Dr. David A. Wheeler
2017-10-22
Outline

• Get a raise!
• Failure example
• **Attack surface**: Where are the inputs?
• Non-bypassability, whitelist not blacklist
• Channels (Sources of input)
• Input data types & non-text validation methods
• Background on text
  – Character names, character encoding, globbing
• Regular expressions for validating strings
• Other notes
Get a raise!

• A fall 2011 student got a raise
  – For securing a key program at his organization
  – Primarily by applying this lecture’s material
    • Aggressively added input validation of untrusted input
Abstract view of a program

Program

Process Data
(Structured Program Internals)

Input

You are here

Output

Call-out to other programs
(also consider input & output issues)
Failure Example: PHF

• White pages directory service program
  – Distributed with NCSA and Apache web servers

• Version up to NCSA/1.5a and apache/1.0.5 vulnerable to an invalid input attack

• Impact: Untrusted users could execute arbitrary commands at the privilege level that the web server is executing at

• Example URL illustrating attack
  – http://webservcer/cgi-bin/phf?Qalias=x%0a/bin/cat%20/etc/passwd

Credit: Ronald W. Ritchey
PHF Coding problems

- Uses `popen` command to execute shell command
- User input is part of the input to the `popen` command argument
- Does not properly check for invalid user input
- Attempts to strip out bad characters using the `escape_shell_cmd` function but this function is flawed. It does not strip out newline characters.
- By appending an encoded newline plus a shell command to an input field, an attacker can get the command executed by the web server

Credit: Ronald W. Ritchey
strcpy(commandstr, "/usr/local/bin/ph -m ");
if (strlen(serverstr)) {
    strcat(commandstr, " -s ");
    escape_shell_cmd(serverstr);
    strcat(commandstr, serverstr);
    strcat(commandstr, " ");
}
escape_shell_cmd(typestr);
strcat(commandstr, typestr);
if (atleastonereturn) {
    escape_shell_cmd(returnstr);
    strcat(commandstr, returnstr);
}
printf("%s%c", commandstr, LF);
printf("<PRE>%c", LF);

phpf = popen(commandstr,"r");
send_fd(phpf, stdout);
printf("</PRE>%c", LF);

Dangerous routine to use with user data
void escape_shell_cmd(char *cmd) {
    register int x, y, l;
    l = strlen(cmd);
    for (x = 0; cmd[x]; x++) {
        if (ind("&;`'\"|?~<>^()[]{}$\", cmd[x]) != -1) {
            for (y = l + 1; y > x; y--)
                cmd[y] = cmd[y - 1];
            l++;
            /* length has been increased */
            cmd[x] = '\';
            x++;
            /* skip the character */
        }
    }
}
**Attack Surface**

- Attacker can attack using channels (e.g., ports, sockets), invoke methods (e.g., API), & sent data items (input strings & indirectly via persistent data).
- A system’s attack surface is the subset of the system’s resources (channels, methods, and data) [that can be] used in attacks on the system.
- Larger attack surface = likely easier to exploit & more damage.

*From An Attack Surface Metric, Pratyusa K. Manadhata, CMU-CS-08-152, November 2008*
Attack Surface: What should a defender do?

• Make attack surface as small as possible
  – Disable channels (e.g., ports) and methods (APIs)
  – Prevent access to them by attackers (firewall, access control)

• Make sure you know every system entry point
  – Network: Scan system to make sure

• For the remaining surface, as soon as possible:
  – Ensure it’s authenticated & authorized (if appropriate)
  – Ensure that all untrusted input is valid (input filtering)
    • Untrusted input = Any input from a source not totally trusted
    • Failures here are CWE-20: Improper Input Validation
  – Many would argue “validate all input”, not just untrusted
    • Trusted admins make mistakes too!

*Input validation of all untrusted inputs* is vital – it helps counter many attacks.
Dividing Up System

• One technique to counter attacks is to divide system into smaller components
  – Smaller components that do not fully trust another
  – Each smaller component has an attack surface

• Thus, even in web applications:
  – Processes might be invoked by an attacker
  – You might have a process that has different privileges

• Design material will discuss further
Examples of Potential Channels (Sources of Input)

- Command line
- Environment Variables
- File Descriptors
- File Names
- File Contents (indirect?)
- Web-Based Application Inputs: URL, POST, etc.
- Other Inputs
  - Database systems & other external services
  - Registry/system property
  - ...

Which sources of input matter depend on the kind of application, application environment, etc. What follows are potential channels. This is *not* a complete enumerated list, these are only examples. You must do input validation of *all* channels where untrusted data comes from (at least).
Discussion: Input sources

• For different kinds of programs:
  – Identify some potential input channels (e.g., ports) and methods (APIs)
    • Do not limit to intended channels & methods
  – What might an attacker try to do?
  – Consider the many different kinds of systems / environments / platforms (e.g., mobile app, web application, embedded device)

• How can you discover “previously unknown” input sources?
Command line arguments

• Command line programs can take arguments
  – GUI/web-based applications often built on command line programs

• Setuid/setgid program’s command line data is provided by an untrusted user
  – Can be set to nearly anything via execve(3) etc., including with newlines, etc. (ends in \0)
  – Setuid/setgid program must defend itself

• Do not trust the name of the program reported by command line argument zero
  – Attacker can set it to any value including NULL
Environment Variables

• Environment Variables
  – In some circumstances, attackers can control environment variables (e.g., setuid & setgid)
  – Makes a good example of the kinds of issues you need to address if an attacker can control something

• If an attacker can control them
  – Some Environment Variables are Dangerous
  – Environment Variable Storage Format is Dangerous
  – The Solution - Extract and Erase
Environment variables: Background

• Normally inherited from parent process, transitively
  – Useful for general environment info
• Calling program can override any environmental settings passed to called program
  – Big problem if called program has different privileges (e.g., setuid/setgid)
  – Without special measures, an invoked privileged program can call a third program & pass to the third program potentially dangerous environment variables
Dangerous Environment Variables

• Many libraries and programs are controlled by environment variables
  – Often obscure, subtle, or undocumented

• Example: IFS
  – Used by Unix/Linux shell to determine which characters separate command line arguments
  – If rule forbid spaces, but attacker could control IFS, an attacker could set IFS to include Q & send “rmQ-RQ*”
  – Well-documented, standard... but obscure
Path Manipulation

• PATH sets directories to search for a command
  ```bash
echo $PATH
/sbin:/usr/sbin:/bin:/usr/bin
  ```

• Attacker can modify path to search in different directories
  `/home/attacker/nastyprograms:/sbin:/usr/sbin:/bin:/usr/bin`

• If the called program calls an external command, attacker can replace the trusted command

• Recommendations:
  – Don’t trust PATH from untrusted source
  – Make “.” (current dir, if there) list after trusted dirs
  – Use full executable name, just in case you forget

Credit: Ronald W. Ritchey
Environment Variable Storage (Normal)

- Environment variables are internally stored as a pointer to an array of pointers to characters
  - `getenv()` & `putenv()` maintain structure

```
ENV

PTR → SHELL = /bin/bash NIL
PTR → HISTSIZE = 10000 NIL
PTR → HOME = root NIL
PTR → LANG = en NIL
NIL
```

Picture by Ronald W. Ritchey
Environment Variable Storage (Abnormal)

- Attackers may be able to create unexpected data formats if can execute directly (e.g., setuid)
  - A program might check one value for validity, but use a different value
  - Environments transitively sent down

```
ENV

PTR  ->  SHELL = /bin/sh NIL

PTR  ->  SHELL = /attack/sh NIL

NIL
```

Picture by Ronald W. Ritchey
Environment variable solution

If attackers might provide environment variable values (setuid or otherwise privileged code), at transition to privilege:

• Determine set of required environmental variables
• Extract their values, and reset or carefully check for validity
• Completely erase environment
• Reset just those environment values
File descriptors

• Object (e.g., integer) reference to an open file
• Unix programs expect a standard set of open file descriptors
  – Standard in (stdin)
  – Standard out (stdout)
  – Standard error (stderr)
• May be attached to the console, or not. A calling program can redirect input and output
  – myprog < infile > outfile
File descriptors

• Don’t assume stdin, stdout, stderr are open if invoked by attacker
• Don’t assume they’re connected to a console
File contents

• Untrusted File - File contents can be modified by untrusted users
  – Including indirectly - can non-trusted users edit it indirectly (e.g., by posting a comment)?
  – Must verify all contents of file before use by trusted program (or handle carefully)

• Trusted File - File contents can’t be modified by untrusted users
  – Must verify that file is not modifiable by non-trusted users
Server-side web applications

• Common Gateway Interface (CGI)
  – Old-but-still-works standard, RFC 3875
  – Server sets certain environment variables influenced by external (usually untrusted) user, e.g., QUERY_STRING
  – Those values need to be validated

• Various web frameworks
  – Enable invoking user-defined scripts/methods
  – Again, must check anything from untrusted user
Some other inputs

• *All* untrusted input that your program must rely on should be carefully checked for validity, and *must* be checked if an attacker can manipulate them. For example:
  – Current Directory
  – Signals
  – Shared memory
  – Pipes
  – IPC
  – Registry
  – External programs (e.g., database systems, other programs on mobile device/server, etc.)
  – Sensors
  – ...

You must do input validation of *all* channels where untrusted data comes from (at least) – not just these!
Non-bypassability

• Make sure attackers cannot bypass checking
  – Find all channels
  – Check all inputs from untrusted sources from them
  – Check as soon as possible

• Client/Server system: Do all security-relevant checking at server in the normal case
  – Client checking can improve user response & lower server load, but...
  – Client checking useless for security
    • Attacker can subvert client or write their own
    • Try to avoid duplicating code using inclusion, etc.
  – Client checking useful to protect against attack from server
Imagine a web application sends this HTML to a web browser as part of a form:

```html
<input name="lastname" type="text" id="lastname" maxlength="100" />
```

Does this HTML provide security-relevant input validation (e.g., to ensure that last names are no more than 100 characters long)?

**NO! THIS DOES NOT PROVIDE ANY SECURITY!**

HTML sent to a web browser is formatted and processed client-side. This makes it trivial to bypass and thus is typically irrelevant for security, e.g., the attacker might write his own web browser client or plug-in. This HTML may be useful to speed non-malicious responses, but it does not counter attack.
Imagine a web application sends this Javascript to a web browser:

```javascript
function regularExpression() {
  var a=null;
  var first = document.forms["form1"]["firstname"].value;
  var firstname_pattern = /^\[A-Z][a-z]{1,30}$/;
  if(first==null || first=="") {
    alert("First name cannot be null");
    return false;
  } else {
    a=first.match(firstname);
    if (a==null || a=="") {
      alert("First name must be of form Xxxxxx");
      return false;
    }
  }
}
```

and also sent this HTML that activated it:

```html
<form action="register.jsp" name="form1" onsubmit="return regularExpression()" method="post">
  <button type="submit"></button>
</form>
```

Does this Javascript provide security-relevant input validation?

**NO! THIS DOES NOT PROVIDE ANY SECURITY!**

Javascript sent to a web browser is executed client-side. This typically makes it trivial to bypass and thus irrelevant for security. This Javascript may be useful to speed non-malicious responses, but it does not counter attack.
Checking the input: Whitelist, **not** blacklist

- Blacklist = pattern that defines all input that shouldn’t be accepted (all other input is accepted)
- Whitelist = pattern that defines all input that should be accepted (all other input rejected)
- A Whitelist or blacklist is a *pattern* or *ruleset* – *not* necessarily a list
- Do **not** implement blacklists for input validation
  - Attackers are clever & can often can find a new “bad” input
  - Users will not warn you that your filter is too loose
- Instead, implement input validation as a *whitelist*
  - Gives little for the attacker to work around
  - If you’re *too* strict, at least the users will tell you
- Blacklist ok if you can provably enumerate (rare!)
- Check *after* decoding (URL decoding, etc.)
  - “abc%20def” == “abc def”

*Use whitelists, not blacklists*
“Blacklists” are useful for testing

• Identify some data you should not accept
  – But don’t use this blacklist as your rule
• Instead, use blacklists to test your whitelist rules
  – I.E., use (subset of) a blacklist as test cases
  – To ensure your whitelist rules won’t accept them
• In general, regression tests should check that “forbidden actions” are actually forbidden
  – Apple iOS’s “goto fail” vulnerability (CVE-2014-1266): its SSL/TLS implementation accepted valid certificates (good) and invalid certificates (bad). No one tested it with invalid certificates!
Input types

- Numbers
- Strings
Numbers

• Check value after converting to a number
  – Number overflow: On a 64-bit machine, usually 18446744073709551615 (2^64-1) \( \rightarrow -1 \)
• Check for min (0? 1? Negative?) & max
  – Make sure all values in range ok (avoid /0)
  – For non-negative integer, use an unsigned integer type
  – Prevent being “too large” for rest of system
  – Note that “only 1 through 100” is a whitelist
• Fractions allowed? If not, use integer type
• If floating point: Watch out for weird cases such as NaN, Infinity, negative 0, under/overflow, etc.
Strings

• Where possible, have an enumerated list
  – Then make sure it is *only exactly* one of those values
  – Could convert to a number

• Otherwise:
  – Limit max length (buffer size & counter DoS)
  – Check that it meets whitelist rule
    • “Correct input always conforms to this pattern”
    • If common type (email address, URL, etc.), reuse rule
    • If very complex, can use compilation tools/BNF
      – More complicated, make sure tools can handle attacks
    • Common tool: Regular expressions (REs)
    • Need background first: char names, encoding, Unicode, globbing
### Common Information Technology Names of Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Common IT Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>bang, &lt;exclamation-mark&gt;, exclamation point</td>
</tr>
<tr>
<td>#</td>
<td>hash, octothorpe, &lt;number-sign&gt; (Warning: “pound” can mean £)</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote, &lt;quotation-mark&gt;</td>
</tr>
<tr>
<td>’</td>
<td>single quote, &lt;apostrophe&gt;</td>
</tr>
<tr>
<td>`</td>
<td>backquote, &lt;grave-accent&gt;</td>
</tr>
<tr>
<td>$</td>
<td>dollar, &lt;dollar-sign&gt;</td>
</tr>
<tr>
<td>&amp;</td>
<td>&lt;ampersand&gt;, amper; amp; and</td>
</tr>
<tr>
<td>*</td>
<td>star, splat, &lt;asterisk&gt;</td>
</tr>
<tr>
<td>+</td>
<td>&lt;plus&gt;</td>
</tr>
<tr>
<td>,</td>
<td>&lt;comma&gt;</td>
</tr>
<tr>
<td>-</td>
<td>dash, &lt;hyphen&gt;</td>
</tr>
<tr>
<td>.</td>
<td>dot, &lt;period&gt;</td>
</tr>
</tbody>
</table>

- Need names to talk about things
- &lt;formal-name&gt; per POSIX 2008
- Used often → few syllables
### Common Information Technology Names of Characters (2)

<table>
<thead>
<tr>
<th>Character</th>
<th>Common IT Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>&lt;slash&gt;, &lt;solidus&gt;</td>
</tr>
<tr>
<td>\</td>
<td>&lt;backslash&gt;</td>
</tr>
<tr>
<td>?</td>
<td>question, &lt;question-mark&gt;, ques</td>
</tr>
<tr>
<td>^</td>
<td>hat, caret, &lt;circumflex&gt;</td>
</tr>
<tr>
<td>_</td>
<td>&lt;underline&gt;, underscore, underbar, under</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>( ... )</td>
<td>open/close, left/right, o/c paren(theses), &lt;left/right-parenthesis&gt;</td>
</tr>
<tr>
<td>&lt; ... &gt;</td>
<td>less/greater than, l/r angle (bracket), &lt;less/greater-than-sign&gt;</td>
</tr>
<tr>
<td>[ ... ]</td>
<td>l/r (square) bracket, &lt;left/right-square-bracket&gt;</td>
</tr>
<tr>
<td>{ ... }</td>
<td>o/c (curly) brace, l/r (curly) brace, &lt;left/right-brace&gt;</td>
</tr>
</tbody>
</table>

Source: The Jargon File, entry “ASCII”. Some entries omitted. Reordered to show contrasts. There programming terms for some character sequences, too, e.g.: 

`<=> (spaceship)`
Character encodings: General

• Characters are represented by numbers
• ASCII common in US
  – 7-bit code, e.g., “A” = 65, “a” = 97
  – *Cannot* represent most other languages
• ISO/IEC 8859-1: 8-bit, most Western Europe
• Windows-1252: 8-bit, like 8859-1 but not
• Other languages have other encodings
  – Must know which encoding for a given document
  – Difficult to handle multiple languages
  – Big mess – we need a single standard for everyone!
Solution: ISO/IEC 10646 / Unicode

• Solution: ISO/IEC 10646 / Unicode
• Defines a “Universal Character Set (UCS)” that assigns a unique number (“code point”) for every “character”
  – ASCII is a subset, so “A” = 65 here too
  – Sometimes different glyphs are considered same character (Han unification of Chinese characters)
  – Sometimes different characters may have identical glyphs (e.g., Cyrillic, Greek, Latin)
  – Once thought 16 bits would be enough – WRONG (changed 1996)
  – Now 21-bit code (including unassigned code points), hex 0…10FFFF
• Defines encodings for how those numbers can be transmitted in a string of bytes
  – UTF-8, UTF-16 (BE/LE/unmarked), UTF-32 (BE/LE/unmarked)
  – Before accepting data, check if valid for that encoding

For more info, see: http://www.unicode.org/faq/
Character encoding: UTF-32

- 32 bits/character, one after the other
- Good news: Every character takes the same amount of space (good for random access)
- Bad news: Big-endian/little-endian (BE/LE)
  - 4 bytes: Does big or little part come first?
  - Fundamentally two UTF-32s: UTF-32BE and UTF-32LE
  - If unmarked, prefix “byte order mark” (BOM) U+FFFE
  - Complicates string concatenation
- Bad news: Lots of wasted space
- Validity check: Each character in range 0…10FFFF
- Used... but not that widely
**Character encoding: UTF-16**

- Sends as a stream of 16-bit values
  - For characters $\leq 2^{16}$, just the character value
  - For other characters, 2 16-bit pairs
- Easier on systems that assumed “16 bits ought to be good enough”: Windows API, Java
  - But a 16-bit “character” might only be part of one, and often people don’t handle this properly
- “Random” access harder, but usually that’s okay
- Less wasted space than UTF-32, more space than UTF-8
- Bad news: Big endian/little endian again
  - Prefix BOM to identify
  - Complicates string concatenation
Character encoding: UTF-8

• Sends characters as a clever 8-bit stream
  – Variable number of bytes, 1-4/character
  – If ASCII, it’s unchanged, so it’s compatible with many existing programs (WIN!)
  – *No* endianness issue, “just works”
    • Easy copy-and-paste to create longer strings
  – Self-synchronizing – easy to find next/previous character

• This is a great encoding!
  – Use it by default if there’s no reason to do otherwise
  – Most common encoding on web [Unicode]
# How UTF-8 Works

<table>
<thead>
<tr>
<th>Code point range</th>
<th>Binary code point</th>
<th>UTF-8 bytes</th>
<th>Example (Source: Wikipedia UTF-8 article)</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0000 to U+007F</td>
<td>0xxxxxxx</td>
<td>0xxxxxxx</td>
<td>character '§' = code point U+0024 = 00100100 → 00100100 → hex 24</td>
</tr>
<tr>
<td>U+0080 to U+07FF</td>
<td>00000yyy yyxxxxxx</td>
<td>110yyyyy 10xxxxx</td>
<td>character '¢' = code point U+00A2 = 00000000 10100010 → 11000010 10100010 → hex C2 A2</td>
</tr>
<tr>
<td>U+0800 to U+FFFF</td>
<td>zzzzyyyy yyxxxxxx</td>
<td>1110zzzz 10yyyyy 10xxxxx</td>
<td>character '€' = code point U+20AC = 00100000 10101100 → 11100010 10000010 10101100 → hexadecimal E2 82 AC</td>
</tr>
<tr>
<td>U+010000 to U+10FFFF</td>
<td>000wwwzz zzzzyyyy yyxxxxxx</td>
<td>11110www 10zzzzz 10yyyyy 10xxxxx</td>
<td>character '筱' = code point U+024B62 = 00000010 01001011 01100010 → 11110000 10100100 10100100 10101101 10100010 → hex F0 A4 AD A2</td>
</tr>
</tbody>
</table>
UTF-8 illegal sequences

• But: Some byte sequences are illegal/overlong
• Before accepting a UTF-8 sequence, check if valid
  – You should check validity for others too, but esp. important UTF-8
  – C0 80 isn’t valid, but is a common representation of byte 0. Think!
• Unchecked invalid sequence might be interpreted as NIL, newline, slash, etc., by your decoder
  – Attacker may be able to bypass your checking if that happens!
Locale

• Locale defines user’s language, country/region, user interface preferences, and probably character encoding
  – E.G., on Unix/Linux, Australian English with UTF-8 is en_AU.UTF-8
• Can affect how characters are interpreted
  – Collation (sorting) order
  – Character classification (what’s a “letter”?)
  – Case conversion (what’s upper/lower case of a character?)
• “POSIX” or “C” locale – often safer, but not always what the user wanted
Visual Spoofing

- Visual spoofing = 2 different strings mistaken as same by user
- Mixed-script, e.g., Greek omicron & Latin “o”
- Same-script
  - “-” Hyphen-minus U+002D vs. hyphen “-” U+2010
  - “ƶ” may be U+007A U+0335 (z + combining short stroke overlay) or U+01B6
- Bidirectional Text Spoofing

For more information on Unicode-related security issues, see:
Conclusions

• Identify/minimize attack surface
  – Where can all untrusted inputs enter?

• Validate **all** untrusted input (non-bypassable)
  – Untrusted = not totally trusted. Might check trusted input too!
  – Use whitelists, **not** blacklists
  – Be maximally strict
  – Numbers: Convert to number, check min/max, use right type
  – Text: Enumerate if you can, reuse checks if you can, in most other cases create limiting RE

• REs often a useful tool for input validation (not only way)
  – Quick (in development time), easy to use, widely available

• Input validation doesn’t make software secure by itself
  – Input validation helps counters many attacks and is a key part