OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Secure Programming

Input Validation

Prof. Dr. Christoph Skornia
christoph.skornia@hs-regensburg.de

Vulnerabilities (revisited)

typical idea of hacking: use malformed input to produce unwanted results

- Top 25 Most Dangerous Software Errors http://cwe.mitre.org/top25/

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Cri...
[6]	76.8	CWE-862	Mi...
[7]	75.0	CWE-...	...e of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)

Insufficient Input Validation

Input Validation

in a lot of computer-languages memory management is as intelligent as a glass. It does not check what and how much you fill into it

Validating input is a programmers task

Excuses strictly forbidden

main problem: wrong assumptions like

- library functions are safe
- colleagues checked input already
- Input is only forwarded to another secure application
- http://danuxx.blogspot.de/2013/03/unauthorized-access-bypassing-php-strcmp.html

Input Validation

Safe assumptions are essential for security:

- Assume all input is guilty until proven otherwise
- Prefer rejecting data to filtering data
- Perform data validation both at input points and at the component level
- Do not accept commands from the user unless you parse them yourself
- Beware of special commands, characters, and quoting
- Make policy decisions based on a "default deny" rule
- You can look for a quoting mechanism, but know how to use it properly
- When designing your own quoting mechanisms, do not allow escapes
- The better you understand the data, the better you can filter it

Input Validation

Examples for regularly requested input:

- File- and Pathnames
  e.g. “Safe File to…”
- URLs
- Environment Variables
- Database entries
  e.g. name, id, size, credit-card-number etc.
- Strings
  (whatever this might be)
- Integers
  (where do natural numbers start… 0 or 1?)

Input Validation

What to do with the input?

Rules of Thumb:

1. Define what you expect !!!
2. Canonicalize (Separation of concerns)
3. Control that given input meets your expectations
4. If it does not meet your expectations check if you can manipulate it (e.g. by cutting to an expected size). If not, drop input.

Input Validation

1.Define what you expect !!!

- What is the purpose of the requested input?
- What format is reasonable (number, strings, telephone number, zip-code etc.)?
- What sizes are reasonable (how long can a URL, a name, a size of shoe be at max)?
- What kind of type conversion contains your code for the requested input (e.g. int g char g char16_t g g...)?

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

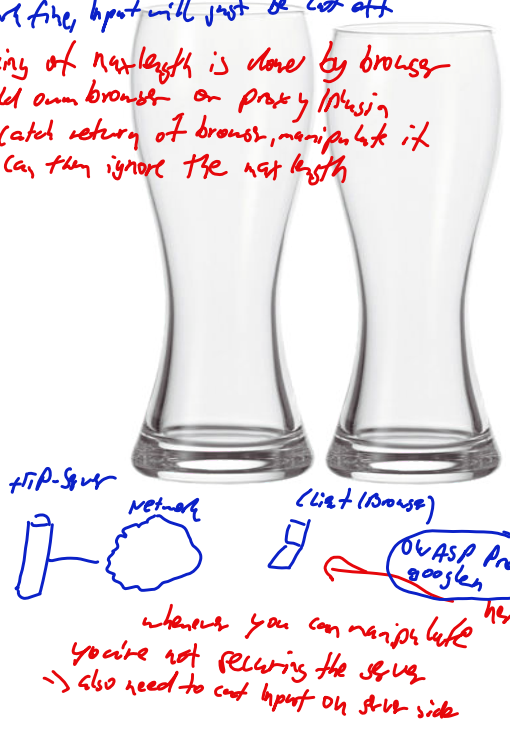
Take Care: Distributed Systems

What's the problem with:

How could an attacker circumvent this code? By accident-errors will work fine, input will just be cut off

```
<form action="demo_form.asp">
  Username: <input type="text" name="usrname"
maxlength="10"><br>
  <input type="submit" value="Submit">
</form>
```

checking of maxlength is done by browser
→ build own browser or proxy (Burp)
⇒ catch return of browser, manipulate it
can then ignore the maxlength



Limit Input Size always also where Input is processed…

e.g.

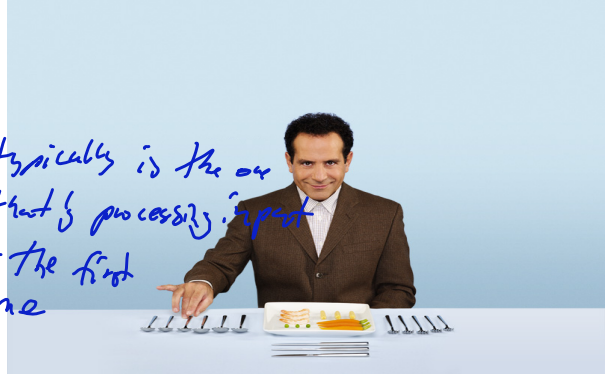
```
if(strlen($addres) > 100) {
                    //Something weird
        } else {
                    //We're good
        }
```

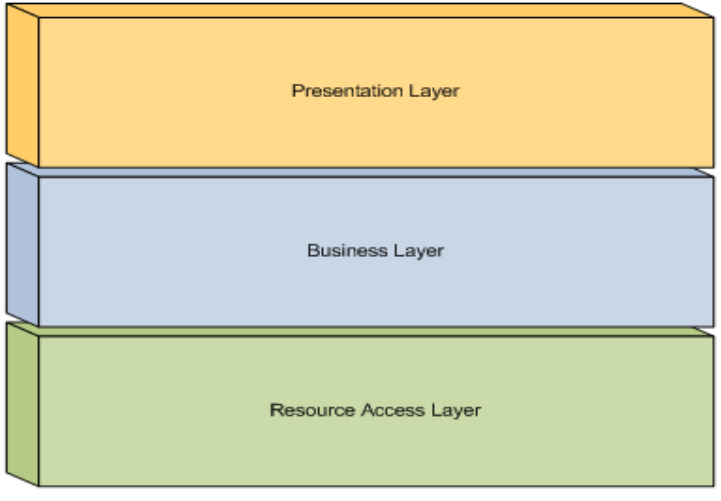
Input Validation

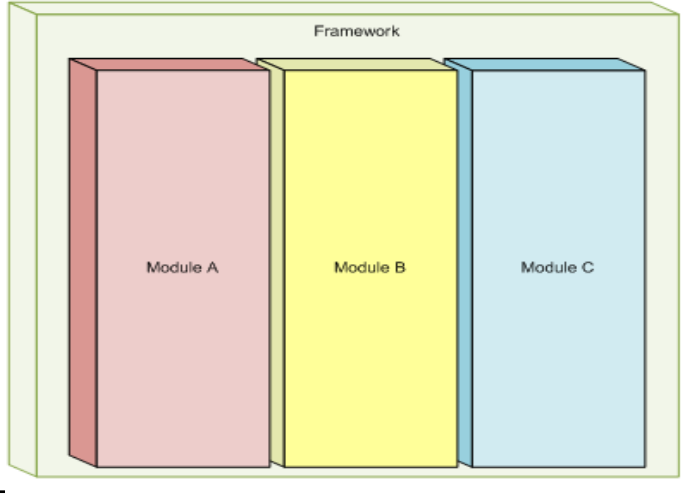
2.Canonicalize

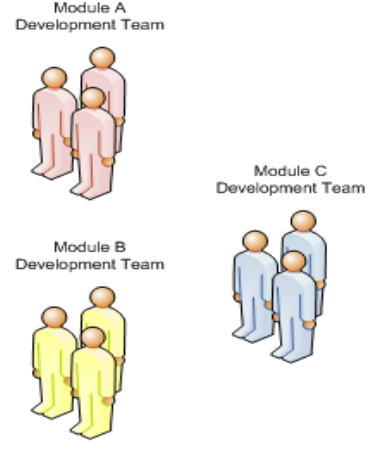
- Separation of concerns:
  - design principle
  - separating a computer program into distinct features that overlap in functionality as little as possible

ask yourself: which module is responsible for canonicalization? typically is the one that is processing input for the first time









Input Validation

typical linux-function for this is realpath

- Filenames and Paths on Unix

```
char *realpath(
        const char *pathname,
        char resolved_path[MAXPATHLEN]
);
```

gives back file-& pathname in absolute path name
→ protects against Past-traversal-attacks

- pathname:
  Path to be resolved
- resolved_path:
  Buffer into which the resolved path will be written. It must be at least MAXPATHLEN bytes in size. realpath( ) will never write more than that into the buffer, including the NULL-terminating byte.
- If the function fails for any reason, the return value will be NULL, and errno will contain an error code indicating the reason for the failure. If the function is successful, a pointer to resolved_path will be returned.

Input Validation

- Filenames and Paths on Windows

```
DWORD WINAPI GetFullPathName(
  _In_   LPCTSTR lpFileName,
  _In_   DWORD nBufferLength,
  _Out_  LPTSTR lpBuffer,
  _Out_  LPTSTR *lpFilePart
);
```

- lpFileName
  Path to be resolved.
- nBufferLength
  Size of the buffer, in characters, into which the resolved path will be written.
- lpBuffer
  Buffer into which the resolved path will be written.
- lpFilePart
  Pointer into lpBuffer that points to the filename portion of the resolved path.
- GetFullPathName( ) will set this pointer on return if it is successful in resolving the path.

When you initially call GetFullPathName( ), you should specifiy NULL for lpBuffer, and 0 for nBufferLength. When you do this, the return value from GetFullPathName( ) will be the number of characters required to hold the resolved path. After you allocate the necessary buffer space, call GetFullPathName( ) again with nBufferLength and lpBuffer filled in appropriately.

Input Validation

Note:

- GetFullPathName( ) requires the length of the buffer to be specified **in characters, not bytes**
- If an error occurs in resolving the path, GetFullPathName( ) will return 0, and you can call GetLastError( ) to determine the cause of the error; otherwise, it will return the number of characters written into lpBuffer.

```
#include <windows.h>

LPTSTR ResolvePath(LPCTSTR lpFileName) {
    DWORD  dwLastError, nBufferLength;
    LPTSTR lpBuffer, lpFilePart;

    if (!(nBufferLength = GetFullPathName(lpFileName, 0, 0, &lpFilePart))) return 0;
    if (!(lpBuffer = (LPTSTR)LocalAlloc(LMEM_FIXED, sizeof(TCHAR) * nBufferLength)))
        return 0;
    if (!GetFullPathName(lpFileName, nBufferLength, lpBuffer, &lpFilePart)) {
        dwLastError = GetLastError(  );
        LocalFree(lpBuffer);
        SetLastError(dwLastError);
        return 0;
    }

    return lpBuffer;
}
```

Input Validation

- URLs

not well-defined; typically only work with ASCII characters

  - RFC 1738 defines the syntax for URLs (Berners-Lee, Masinter & McCahill, Dec 1994)
  - URLs are written as follows: <scheme>:<scheme-specific-part>
    - Scheme names consist of a sequence of characters.
      - The lower case letters "a"--"z",
      - digits, and the characters
      - plus ("+"),
      - period ("."), and
      - hyphen ("-") are allowed.

need to canonicalize URLs (eliminate ambiguity)

Chromium good but complex

can attack server with malformed URL

  - URLs are written only with the graphic printable characters of the US-ASCII coded character set. The octets 80-FF hexadecimal are not used in US-ASCII, and the octets 00-1F and 7F hexadecimal represent control characters; these must be encoded.

Input Validation

- safe and unsafe characters
  - space character is unsafe
    because significant spaces may disappear and insignificant spaces may be introduced when URLs are transcribed or typeset or subjected to the treatment of word-processing programs.
  - characters "<" and ">" are unsafe
    because they are used as the delimiters around URLs in free text; the quote mark (""") is used to delimit URLs in some systems.
  - character "#" is unsafe
    because it is used in World Wide Web and in other systems to delimit a URL from a fragment/anchor identifier that might follow it.
  - character "%" is unsafe because it is used for encodings of other characters.
  - characters "{", "}", "|", "\", "^", "~", "[", "]", and "`" are unsafe because gateways and other transport agents are known to sometimes modify such characters.
- All unsafe characters must always be encoded within a URL. For example, the character "#" must be encoded within URLs even in systems that do not normally deal with fragment or anchor identifiers, so that if the URL is copied into another system that does use them, it will not be necessary to change the URL encoding.

Input Validation

- Security Problems:
  - multiple encodings
    - example:
      - %xx (with xx being digits) will be encoded, but repeated appearance of such combination might lead to repeated encoding.
      - "%25" will be encoded to "%"
      - "%34" will be encoded to "4"
      - "%31" will be encoded to "1"
      - a "%25%34%31" will be encoded to "%41"
      - "%41" will be encoded to "A"
    - Conclusion: encoding multiple times lead to ambiguities in URLs
    - ambiguity lead to potentially undefined states!
    - How often to encode or decode?
    - even worse: you end up with certain sequences of characters that are impossible to represent

even different outputs for each System
-> canonicalization on server-side; on client-side could be misinterpreted

Input Validation

- Security Problems:
  - NULL-termination of C-style strings
    (btw: strings is a topic on its own, which will be treated later)
    - NULL-terminator can be encoded anywhere in the URL#
  - Path traversals
    - What happens if you enter:
      http://my.server.net/../../secret-document.pdf?
    - How to get rid of that?

Input Validation

- Solution:
  - manual:
    - read http://tools.ietf.org/html/rfc3986 carefully and develop your own normalization code (no serious alternative)
    - file://localhost/Volumes/UserData/skc39497/Documents/HSR/Vorlesungen/Secure-Programming/Materialien/LeKi05a.pdf
    - http://dev.w3.org/html5/spec-preview/urls.html#absolute-url
  - predefined functions available:
    - Windows UrlCanonicalize ()
    - multiple systems: Google: Chromium project contains very advanced URL canonicalization code

Regular Expressions

- IEEE Std 1003.2 ("POSIX.2“)
  - C, C++ (wenn Sie boost sowieso verwenden dann besser Boost.Regex) ansonsten

```
#include <regex.h>
int regcomp(regex_t * restrict pregconst char * restrict patternint cflags);

int regexec(const regex_t * restrict pregconst char * restrict string, size_t
nmatchregmatch_t pmatch[restrict]int eflags);

size_t regerror(int errcodeconst regex_t * restrict preg, char * restrict
errbufsize_t errbuf_size);

void regfree(regex_t *preg);
```

Regular Expressions

- Perl

```
$string =~ m/pattern/modifier
$string =~ s/pattern/replace/flags
```

- PHP

```
preg_match('/pattern/flags', $string)
preg_replace('/pattern/flags', 'replace', $string)
```

- Ruby

```
string.match(/pattern/flags) or string.match(/pattern/flags) {|match|...} #
string.sub(/pattern/flags, replace) or string.sub(/pattern/flags) {|replace|...}
string.gsub(/pattern/flags, replace) or string.gsub(/pattern/flags) {|replace|...}
```

thanks for your interest

to be continued