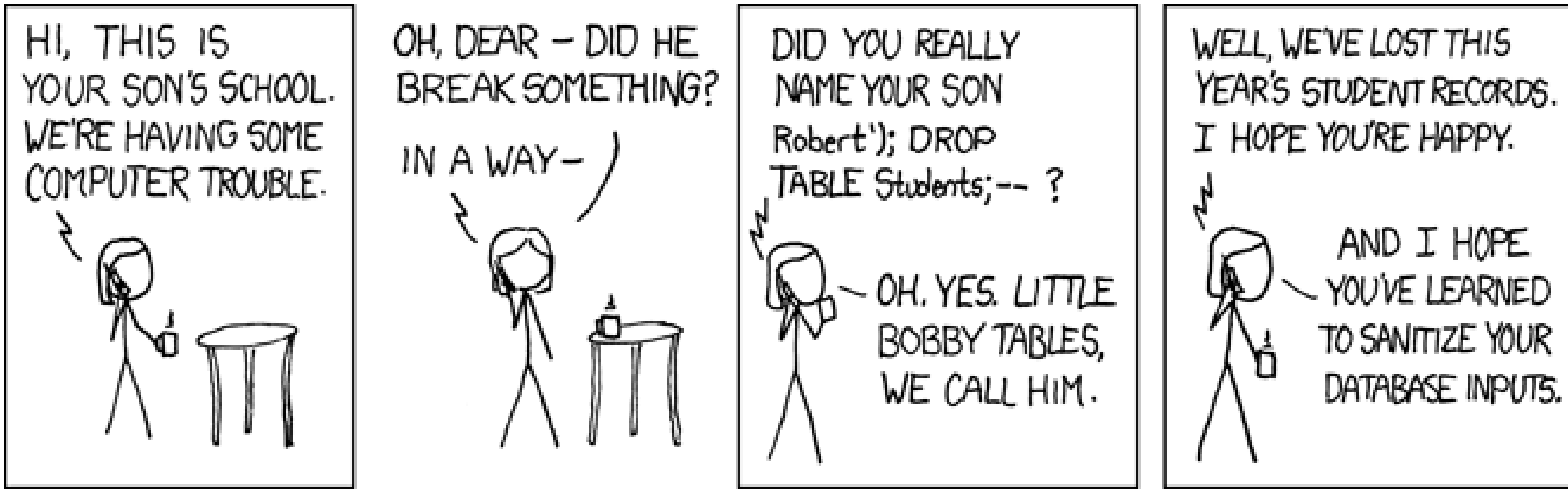
Input Validation

Thomas L. "Trey" Jones, CISSP, CEH

Exploits of a Mom (from xkcd)



“Never Trust Input”

- Un-validated Input forms the basis for some of the worst and most frequently exploited vulnerabilities
  - Buffer Overflows
  - Integer Overflows
  - Format String
  - Injection Flaws
    - Command
    - SQL
    - LDAP

What to validate

- Validate all input.
- Validate input from all sources.
- Establish trust boundaries.
  - Store trusted and untrusted data separately to ensure that input validation is always performed.

Validate all input

- Validate input even if it:
  - Is delivered over a secure connection,
  - Arrives from a "trusted" source, or
  - Is protected by strict file permissions
  - The program is accessed by only trusted users.
- Two major groups:
  - Syntax checks that test the format of the input
  - Semantic checks that determine whether the input is appropriate

The collection of places where an application accepts input can loosely be termed the application's attack surface [Howard and LeBlanc, 2002]

Validate Input from All Sources

- Perform input validation on user input and on data from any source outside your code.
  - Command-line parameters
  - Configuration files
  - Data retrieved from a database
  - Environment variables
  - Network services
  - Registry values
  - System properties
  - Temporary files
  - etc.

Configuration Files

- Version 1.3.29 of Apache's mod_regex and mod_rewrite modules

The kind of input the program expects:

```
RewriteRule ^/img(.*) /var/www/img$1
```

Input that causes a buffer overflow:

```
RewriteRule
^/img(.)(.)(.)(.)(.)(.)(.)(.)(.)(.*) \

/var/www/img$1$2$3$4$5$6$7$8$9$10
```

The Culprit Code

```
int ap_regexec(const regex_t *preg, const char *string,
size_t nmatch,
               regmatch_t pmatch[], int eflags);
typedef struct backrefinfo {
  char *source;
  int nsub;
  regmatch_t regmatch[10];
} backrefinfo;
...
else {  /* it is really a regexp pattern, so apply it */
  rc = (ap_regexec(p->regexp, input,
         p->regexp->re_nsub+1, regmatch, 0) == 0);
```

Correct Version

```
typedef struct backrefinfo {
  char *source;
  int nsub;
  regmatch_t regmatch[AP_MAX_REG_MATCH];
} backrefinfo;
...
  else {  /* it is really a regexp pattern, so apply it
*/
  rc = (ap_regexec(p->regexp, input,
        AP_MAX_REG_MATCH, regmatch,0) == 0);
```

Injection Flaws

- Involves the insertion of control structures from user input where the program was expecting data.
- All major scripting and markup languages are potentially vulnerable.
- Attacks can involve unauthorized disclosure of sensitive information, authentication bypass, arbitrary code execution, data loss, and more.

Injection

(adjective) (name) sat on a (thing).

(adjective) (name) had a great fall.

All the King's (plural things) and all the King's (plural things) couldn't put (adjective) (name) back together again.

Injection

- adjective = "Humpty"
- name = "Dumpty"
- thing = "wall."
- thing, plural = "horses"
- thing, plural = "men"

- Malicious Injection "He reminds me of my last manager"

Injection

Humpty Dumpty sat on a wall. He reminds me of my last manager.

Humpty Dumpty had a great fall.

All the King's horses and all the King's men couldn't put Humpty Dumpty back together again.

Command Injection

- Untrusted data passed through and interpreted as a command
- Unprivileged users given full control of directory structure or unauthorized data access
- Commonly through API calls that directly call the system command interpreter without validation

Affected Languages

- Any language where commands and data are placed inline together
- Most languages handle this vulnerability by providing good APIs with proper input validation
- New APIs can still introduce new command injection errors

Prevention and Countermeasures

- Perform input validation before passing to command processor (Canonicalization of input)
- Fail securely if input validation check fails
  - Signal an error - refuse to run command as is
  - Log the error and all relevant data
- Use a *whitelist* validation approach
  - Use regular expressions to ensure that input contains no dangerous meta-characters, such as “;” or “&&”
- Write your own secure API wrappers
  - Use additional validation techniques
  - Ensures that validation is always performed

Database Queries

- Database must often be granted a level of trust.
  - Generally the database is often the only source of truth.
- Programs that rely on the database should verify that information is well formed and meets reasonable expectations.
- Check that fields contain safe, sane content free from metacharacter attack
- Check for only one row of results if inputs are supposed to yield a unique result.

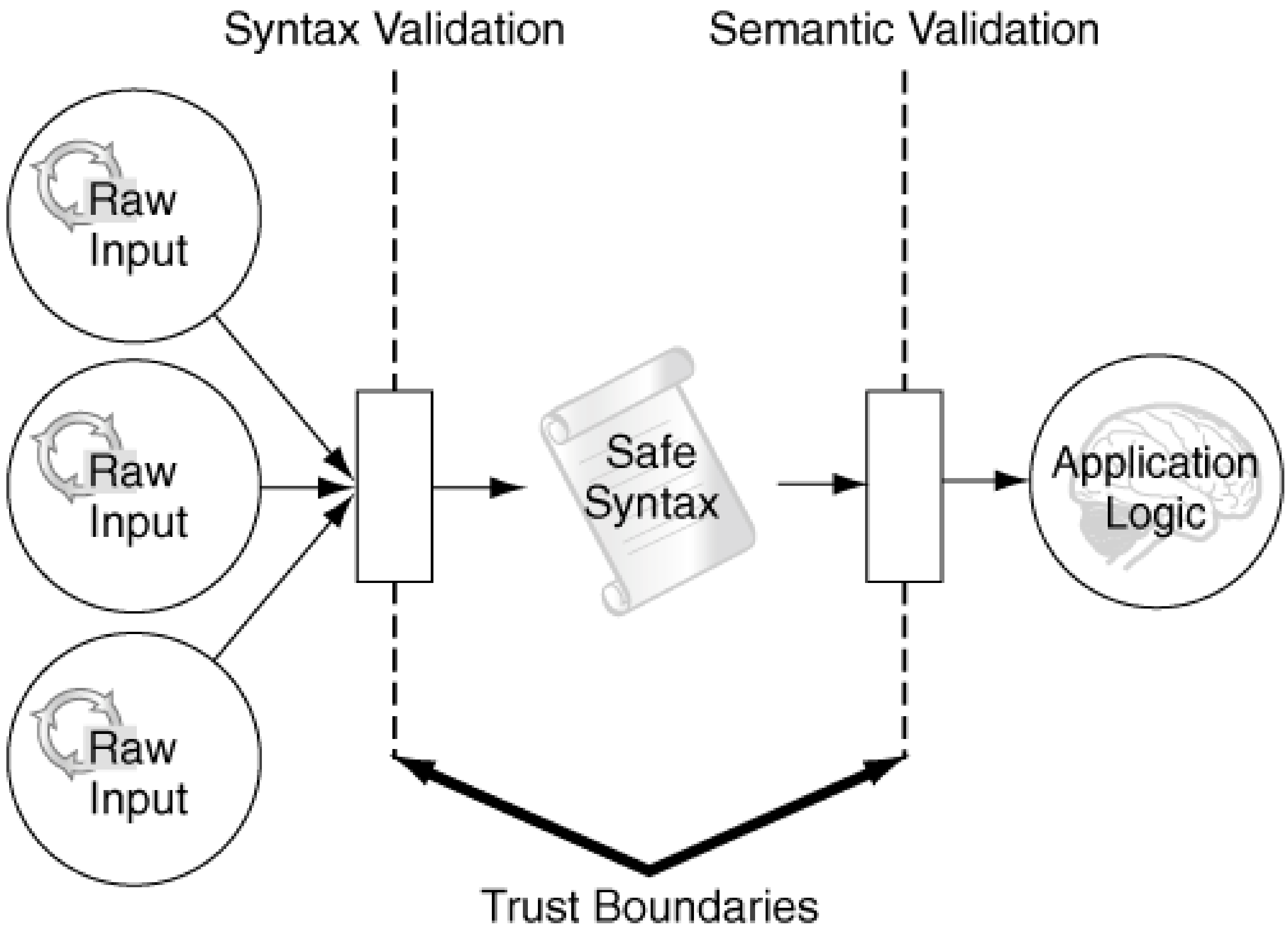
Network Services

- Data coming off the network shouldn't be trusted by default
- Do not rely on DNS names or IP addresses for authentication
  - DNS cache poisoning
  - IP Spoofing

Establish Trust Boundaries

- A trust boundary can be thought of as a line drawn through a program.
  - On one side of the line, data are untrusted.
  - On the other side of the line, data assumed to be safe for some particular operation..
  - Validation logic allows data to cross the trust boundary, to move from untrusted to trusted.

```
// JAVA HTTP Example
status = request.getParameter("status");
if (status != null && status.length() > 0) {
   session.setAttribute("USER_STATUS", status);
}
```


Syntax Validation
Semantic Validation
Raw Input
Raw Input
Raw Input
Safe Syntax
Application Logic
Trust Boundaries

How to Validate

- Use strong input validation.
- Avoid blacklisting.
  - Avoid checking explicitly for bad input: blacklist validation
  - Only accept well-formed input: whitelist validation
    - Regular expressions are your friends
- Don’t mistake usability for security.
- Reject bad data.
  - Don’t try to repair it
- Make good input validation the default.
- Always check input length.
- Bound numeric input.

Use Strong Input Validation

- Indirect Selection
  - Create a list of legitimate values that a user is allowed to specify
  - Allow user to supply only the index into that list
- Check input against a list of known good values
  - Known as Whitelisting
- Do not attempt to check for specific bad values
  - Known as Blacklisting

Why Blacklisting Fails

Why “Blacklisting” Fails

Don’t Mistake Usability for Security

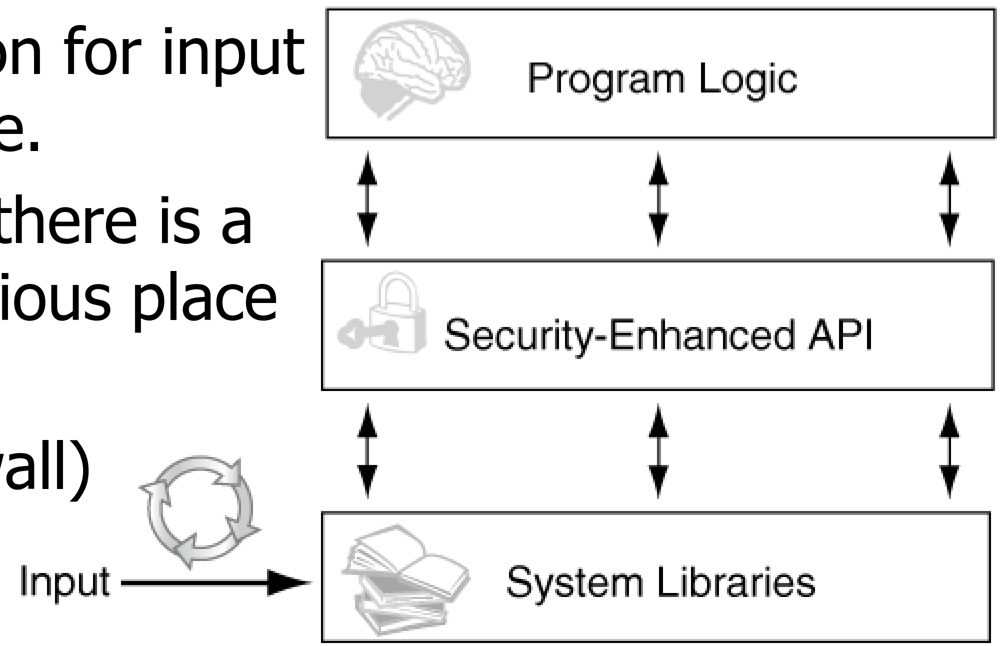
- User-friendly input validation
  - Meant to catch common errors
  - Provide easy-to-understand feedback to legitimate users when they make mistakes.
- Input validation for security purposes
  - Exists to contend with uncommon and unfriendly input.

Reject Bad Data

- Do not repair data that fail input validation checks. Instead, reject the input.

Make Good Input Validation the Default

- Standard methods for accepting input don’t provide a built-in facility for doing input validation.
- Don’t code up new solution for input validation each occurrence.
- Arrange program so that there is a clear, consistent, and obvious place for input validation.
- NOT an Input Filter (firewall)
  - See textbook



Security-Enhanced API's

- A security-enhanced API improves your ability to do the following:
  - Apply context-sensitive input validation consistently to all input.
  - Understand and maintain the input validation logic.
  - Update and modify your approach to input validation consistently.
  - Be constant. If input validation is not the default, it is easy for a developer to forget to do it.
  - See *readlink*() example in textbook
- Must choose the correct set of functions to set on top.

Wrapper to null terminate

```
size_t strlcpy(char *dst, const char *src, size_t siz) {
  char *d = dst;
  const char *s = src;
  size_t n = siz;
  if (n != 0 && --n != 0) {
    do {
      if ((*d++ = *s++) == '\0')
        break;
    } while (--n != 0);
  }
  /* Not enough room in dst, add NULL and traverse rest of src */
  if (n == 0) {
    if (siz != 0)
      *d = '\0';          /* NULL-terminate dst */
    while (*s++);
  }
  return(s - src - 1);     /* count does not include NUL */
}
```

Check Input Length

- Always check input against a minimum and maximum expected length.
  - Length checks don't require much knowledge about the meaning of the input
- Make it harder for an attacker to exploit other vulnerabilities in the system
- Watch out, though—if the program transforms its input before processing it, the input could become longer in the process.

Bound Numeric Input

- Check numeric input against both a maximum value and a minimum value as part of input validation.
- Watch out for operations that might be capable of carrying a number beyond its maximum or minimum value.

Integer Overflows$_1$

- For nearly every binary format available to represent numbers, there are operations that don't give you typical results as you would expect on pencil and paper.
  - Some languages implement range–checked integer types
    - Reduce problems when used consistently
- Occurs when an integer is increased beyond its maximum value and wraps-around or "overflows" into its minimum value.
- Effects range from crashes and logic errors to escalation of privileges and execution of arbitrary code
- Can be triggered by user provided input

Integer Overflows$_2$

- The following operations are likely to cause an integer overflow:
  - Casting operations
  - Operator conversions
  - Arithmetic Operations
  - Comparison Operations
  - Binary Operations

Affected Languages

- All languages are affected by integer overflows
  - Prone to denial of service and logic errors
- Overflows can be signed or unsigned
- C and C++ have true integer types
- C# insists on signed integers
- Java only supports a subset of the full range of integer types
  - Supports 64 bit integers
  - Only supports the char unsigned type

0	0	0
		3

9	9	7
0	0	9

Prevention and Countermeasures

- Check numeric input against a max and min bound before using it, and after any operations which may cause overflow.
- Make checks for integer problems straightforward and easy to understand.
- Use unsigned integers where possible for array offsets and memory allocation sizes.
- Check all calculations used to determine memory allocations or array indexes.
- Pay close attention to code that catches integer exceptions.

Preventing Metacharacter Vulnerabilities

- Allowing attackers to control commands sent to the database, file system, browser, or other subsystems leads to big trouble.
  - SQL Injection
  - Path Manipulation
  - Command Injection
  - Log Forging

Bad Example: Using String Concatenation for Database Queries$_1$

- The following query is constructed by concatenating control structures with user provided input:

```
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ itemName + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

- An attacker can change the meaning of the query by supplying metacharacters in the input.

Bad Example: Using String Concatenation for Database Queries$_2$

- The Programmer intended the query to be as follows:
  - ```
    SELECT * FROM items WHERE owner =
    <userName> AND itemname = <itemName>;
    ```
- An attacker can change the meaning to this:
  - ```
    SELECT * FROM items WHERE owner = 'wiley'
    AND itemname = 'name' OR 'a'='a';
    ```
- Which is equivalent to:
  - ```
    SELECT * FROM items;
    ```
- Now the attacker can see all entries in the items table.
    ```

Solution: Parameterized Queries

- Parameter binding prevents user input from changing the meaning of the statement:

```
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT * FROM items WHERE owner = ?"
+ " AND itemname = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, userName);
stmt.setString(2, itemName);
ResultSet rs = stmt.executeQuery();
```

- The statement is parsed first before parameter substitution occurs.

Bad Example: Command Injection

- The following allows user input to affect the command that is executed:

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

Solution: Command Injection

- The following uses a white list to validate user input:

```
final static int MAXNAME = 50;
final static String FILE_REGEX =
"[a-zA-Z]{1,"+MAXNAME+"}"; // vanilla chars in prefix
final static Pattern BACKUP_PATTERN = Pattern.compile(FILE_REGEX);
public void validateBackupName(String backupname) {
if(backupname == null
|| !BACKUP_PATTERN.matcher(backupname).matches()) {
throw new ValidationException("illegal backupname");
}
}
...
String btype = validateBackupName(request.getParameter("backuptype"));
String cmd = new String("cmd.exe /K \"c:\\util\\rmanDB.bat "
+ btype + "&&c:\\utl\\cleanup.bat\"")
Runtime.getRuntime().exec(cmd);
```

Bad Example: Path Manipulation

- The following allows user input to affect the path to a file being deleted:

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" +
rName);
rFile.delete();
```

Solution: Path Manipulation

- The following uses a white list to validate user input:

```
final static int MAXNAME = 50;
final static int MAXSUFFIX = 5;
final static String FILE_REGEX =
"[a-zA-Z0-9]{1,"+MAXNAME+"}" // vanilla chars in prefix
+ "\\.?" // optional dot
+ "[a-zA-Z0-9]{0,"+MAXSUFFIX+"}"; // optional extension
final static Pattern FILE_PATTERN =
Pattern.compile(FILE_REGEX);
public void validateFilename(String filename) {
if (!FILE_PATTERN.matcher(filename).matches()) {
throw new ValidationException("illegal filename");
}
}
```

Summary

- Identify all the program’s input sources
- Choose the right approach to performing input validation
- Track which input values have been validated and what properties that validation checked
- Keep an eye out for the way different components interpret the data your program pass along

References

[Lavenhar, 2005] Lavenhar, Steven R. *Source Code Analysis Tools – Business Case*. Cigital, 2005

[Viega, 2002] Viega, and Gary McGraw. *Building Secure Software*. 3rd ed. Boston, MA: Addison-Wesley, 2002.

BuildSecurityIn.net Coding Practices
(https://buildsecurityin.us-cert.gov/portal/article/knowledge/Coding_Practices)

BuildSecurityIn.net Coding Rules
(https://buildsecurityin.us-cert.gov/portal/article/knowledge/Coding_Rules)

BuildSecurityIn.net Source Code Analysis Tools
(https://buildsecurityin.us-cert.gov/portal/article/tools/code_analysis)

CERT – Secure Coding
(https://www.cert.org/secure-coding)

BuildSecurityIn.net The Common Criteria
https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/239.html#dsy239_refs