Basic Principles

User Input Is Untrusted

- Can't assume anything about user input
- User may make mistake
- May misunderstand input format
- Input errors may occur
- Or user may be malicious

All Input Must Be Checked

- Any language primitive merely attempts to perform input
- Attempt may succeed or fail (system error, improper format, etc.)
- Must check that:
  - Input was actually read
  - It has the right format

Input Categories and Dangers

- Numbers: invalid, too large, negative, ...
- Strings: control characters, commands
- Filenames: directory traversal, command injection
- URLs: bad site, redirect, parameters, ...
- Anything: too large, malicious encoding, exploit processing algorithm (time/space to transform => denial of service)

Whitelisting, Not Blacklisting

- Fail-safe defaults
- Be absolutely sure that user input is safe
- Establish restrictive set of safe (normal) characters
- Disallow everything not in safe set
- **Do not** rely on identifying “bad” patterns (high risk of forgetting some)

Robust Input Handling

Two Steps of Input Checking

- Any language primitive merely attempts to perform input
- Attempt may succeed or fail (system error, improper format, etc.)
- Must check that:
  1. Input was actually read
  2. It has the right format

How Input Errors Are Signaled

1. **By extending the result type with a value denoting error**
   - Example: reading a char in C

     ```
     int c = getchar();
     ```

- Normal return value is a char (8 bits)
- Special value `EOF` (−1) for **no char read**
  - => **normal char values are converted to unsigned int**

2. **By choosing a special value for error**

   - Example: reading a line in C

     ```
     char buf[80];
     if (fgets(buf, sizeof(buf), stdin))
     ```

   - `fgets` returns either `buf` (read OK) or `NULL` (nothing was read)

3. **By returning an error code**

   - Example: reading with `scanf` in C

     ```
     int x, y;
     if (scanf("%d%d", &x, &y) == 2)
     ```

   - Returns the number of items read, or `EOF`
   - In addition, should check overflow: `errno` variable

4. **By throwing an exception**

   - Example: reading an array with specified number of bytes in Java

---

Regular Expressions for Input Validation

Regular Expression Basics

- Rigorously defined expressions that characterize a family of strings
- Composed by: repetition, concatenation, alternative
- Meta-chars: `?` (optional), `*` (repeat >= 0), `+` (repeat at least once), `|` (alternative)
- Different flavors depending on language
  - **POSIX** extended regular expressions
  - **Perl-compatible** regular expressions

Regular Expression Examples

- Identifiers: `[A-Za-z][A-Za-z0-9]*`
- Can specify place of occurrence:
  - `^` at beginning
  - `$` at end

Use for Validation

- Whitelisting (specify allowed patterns)
  - Example: validating emails, filenames, URLs
- Ensure input is completely specified as valid:
  - don't allow partial validity

Regex Validation Issues

- **CWE-185: Incorrect Regular Expression**
- **it tries to match a phone with a regular expression**

```
$phone = GetPhoneNumber();
if ($phone =~ /\d+-\d+/) {
    system("lookup-phone $phone");
} else {
    error("malformed number!");
}
```

Problem: **it does not check globally, matches anything with a digit-hyphen-digit SUB-STRING**

Definition of Enriched Regular Expression: regular expressions formed with the aid of `+`, `*`, etc.

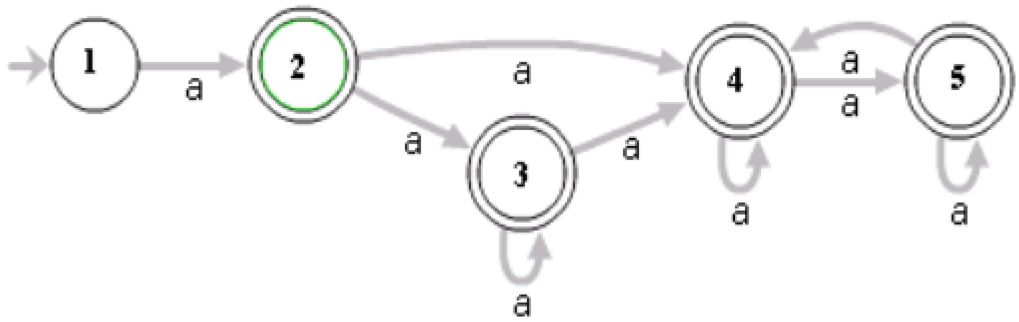
Regular Expression Search in Python

```python
import re

text = "My number is 123-456-7890"
pattern = r"\d{3}-\d{3}-\d{4}"

match = re.search(pattern, text)
if match:
    print("Found:", match.group())
```

```
Found: 123-456-7890
```

Denial of Service (DoS) Attacks

- **Regular expression search ⇒ Non-deterministic automata**
- Bad constructions involve:
  - repetitions of repetitions: `(a+)+`
    - ⇒ it could be written as `(a+)`
  - alternation with overlapping: `(a|aa)+`
    - ⇒ it suffices if we repeat once `(a+)`
  - repetion of repetitions: `([a-zA-Z]+)*`
  - etc.



Denial of Service Attacks

Regular Expression Search :

- **Approach One**:
  1. Building a nondeterministic automaton
  2. trying out many possible paths until finding a path that matches the input string
- **Approach Two**:
  1. Building a nondeterministic automaton
  2. Converting to a deterministic automaton
  - it can be exponential in the size of the regex

Naive non-deterministic constructions, or enriched regular expressions are vulnerable to malicious inputs that take exponential time.

all of these expressions have exponential behaviour for long strings of `aaaaaaaa` with another final char.

ReDoS Attacks

Definition: ReDoS (Regular Expressions Denial of Service attacks)

- Denial of service attacks on regex **libraries** are problematic: widely used on Web.
- Solution: Use known libraries that use deterministic constructions (automata) if possible
- Limit time and resources allocated
- Use tools to detect malicious regexes

Handling File Names

Filename Manipulation Dangers

- Untrusted users may provide filenames that:
  - Break filename processing functions
  - Widen attack on system (access parent directories)
  - Confuse other users by similarity

Directory Traversal

- Including `..` (parent directory) may allow user to specify a file outside the intended directory of access
- Usual solution: prohibit `..`, possibly also directory separators
- Better: limit the set of allowed filename characters

Bad Directory Check

- Check if in one of two allowed directories

```
if(!strncmp(fname,"/usr/lib/safe/",14)) {
  // OK
 }
 else
   if(!strncmp(fname,"/usr/lib/old/",13)) {
     // OK
   }
```

- But `/usr/lib/safe/../../../etc/passwd` escapes the check!

Filename Globbing

- Prohibit using filename wildcards: `*`, `?`, also brackets `[ ]` and braces `{ }`
- Globbing may lead to leaking filenames or processing unintended files
- Globbing may lead to leaking list of filenames matching a pattern
- Or may cause processing of multiple files when a single one was intended

Filename Globbing

- Filename globbing may lead to DoS if resulting set of files is too large `*/../*/../*/../*` ...
- If globbing is needed, limit resources that process can consume (CPU, memory).

Problematic Patterns

- **Leading dash (`-`)** interpreted as a program option
  - Workaround: add `./` to filename or use dash-dash to signal end of option list.
- **Control characters**: may confuse terminal, be confused as separators, break lines
- **Spaces**: confused as multiple arguments. Place filename in doouble quotes `" "`
- **Invalid encodings** (e.g., malformed UTF-8)

- **Shell metacharacters**: `<` `>` `~` etc.

Malicious Truncation Misuse

```
snprintf(buf, sizeof(buf), "/data/profiles/%s.txt", username);
fd = open(buf, O_RDONLY);
```

A long string can a force different extension (e.g., .php, which will be executed by server).

```
snprintf(buf, sizeof(buf), "/data/%s_profiles.txt", username);
fd = open(buf, O_RDONLY);
```

Could open arbitrary files, controlling length with extra `/` or `/./` at start of string.

Character Encoding Issues

The Issue of Encodings

- Software interacts across different components, each with different encoding rules.
- When interfacing, these requirements must be adapted to one another.
- **CWE-150**: Improper Neutralization of Escape, Meta, or Control Sequences.

Canonicalize, Then Validate

- Violation of this principle leads to many security errors
- If data is not canonicalized first, exception to the verification rule might slip in
- **Bad**:

  ```
  path.startsWith("/safedir/")
  ```

- **Good**:

  ```
  f = new File(path);
  f.getCanonicalPath().startsWith("/safedir/")
  ```

Escaping Metacharacters

- Usually done by prefixing with an escape character (e.g., `\`)

- Don't forget escaping the escape char: `\\`
- Example: poor fix against SQL injection, want to avoid username with quote in

```
"WHERE user = '" + $username + "'"
```

- But username `bob\'` is escaped to `bob \\'` and yields the following, where the scaping is applied to the **backslash** symbol!

```
WHERE user = 'bob\\'
```

Auditing Proper Escaping

- Want to check that escaped characters do
- Check where escaped input is decoded
- Check security decisions based on input
- If decision is before the decoding: flaw!
- Example: check for / before hex-decoding pathname: will miss `%2F` encoding for `/`

Unicode-Related Issues

- Homographic attacks: characters with different code but same representation
- Can be used for visual deception/phishing
- Same value can be represented in many ways: with different word size (UTF-8, 16, or 32), switching byte order, etc.
- More problematic if encoding/decoding done several times / in several ways

---

Double Encoding

- Applying encoding twice might bypass improperly designed filters
- Example: avoid detection of `'../'`.

```
enc(enc(../)) = enc(%2e%2e%2f) = %252e%252e%252f
```

- decoded once yields back `%2e%2e%2f` which hides `'../'`
- Or, hide `<script>alert('XSS')</script>` by double-encoding `<` and `/` and `>`

Command Injection: Overview

What is Command Injection?

- CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')
  - CWE-77
- CWE-78: OS Command Injection
  - CWE-78
- Untrusted data used in string executed as a command by the application
- Attacker gains an undesired capability
- Not limited to script/language commands

Examples Seen Before

- Format string vulnerability

```
char s[80];
if (fgets(s, sizeof(s), stdin))
printf(s);
```

- Intent of code: print string s as data
- Problem: %d, %x etc. specifiers have role of ***command***
- **CWE Category**: Tainted input to command

Execute Extra Command (C)

- Intent: display file with given name

```
int main(int argc, char** argv) {
  char cmd[CMD_MAX] = "/usr/bin/cat ";
  strcat(cmd, argv[1]);
  system(cmd);
}
```

- User could add a command after the following commands would executed iin sequence:

```
foo; rm -rf *
```

- There is one more flaw in this code ...

Untrusted Environment (Java)

- Intent: execute initialization script found relative to the application install directory

```
String home=
System.getProperty("APPHOME");
String cmd = home + INITCMD;
java.lang.Runtime.getRuntime().exec(cmd);
```

- If environment variable `APPHOME` set by attacker, `INITCMD` will lead to malicious file.

Flaw in vacation / sendmail

- vacation program designed to send mail on behalf of vacationing user
- Culprit line:

  ```
  execl(_PATH_SENDMAIL, "sendmail", "-f", myname, from, NULL);
  ```

- Parameter myname under attacker control
- Attacker could set to `-C/path/to/file`
- sendmail interprets this as "run command"

OS Command Injection

- Special case, when command target is the operating system
- Two variants, based on intent:
  - Execute specific command + arguments e.g. nslookup hostname (exploitable for malicious hostname)
  - Execute arbitrary command but programmer does not expect this could be attacker-controlled

---

DNS Lookup (Perl)

- Intent: generate HTML with DNS output

```
use CGI qw(:standard);
$name = param('name');
$nslookup = "/path/to/nslookup";
print header;
```

```
if (open($fh,
     "$nslookup $name|")) {
  while (<$fh>) {
    print escapeHTML($_);
    print "<br>\n";
  }
  close($fh);
}
```

DNS Lookup (cont.)

- Attacker provides URL-encoded name:

  ```
  cwe.mitre.org%20%3B%20/bin/ls%20-l
  ```

- Last part, decoded:

  ```
  ; /bin/ls -l
  ```

- Executed command:

  ```
  nslookup cwe.mitre.org;  /bin/ls -l
  ```

- Page will list all files in directory (information disclosure)

Command Injection: Language Specifics

Shell

- Know and check various metacharacters:
  - sequencing `;`
  - background `&`
  - pipe `|`
  - redirect `<` and `>`
  - backquote (evaluate) `` ` ``
  - variable `$`
  - wildcards `*` `?`
  - quotes, whitespace, parantheses ...

C/C++

- Functions that launch a shell with given command line
    - `system()` -- standard library
    - `ShellExecute()` -- on Windows
- All shell metacharacters are interpreted

Java

- `java.lang.Runtime.exec()`
- Better security design than `system()`
- Does not invoke the shell, so no support for shell metacharacters
- Splits argument string into words and launches program given by first word with other arguments
- Still potential for injection, depending on how program interprets arguments

---

Perl

- system() calls shell, interprets metachars
- open() can open both files and processes
- interprets metacharacters at beginning and end of filename
- if vertical bar is first or last, rest of name is command for which input/output is piped
- `eval()` evaluates argument as Perl code
- backticks ` invoke a shell

Python

- `compile()` compiles string into code
- `exec()`, `eval()` evaluates data as code
- `execfile()` parses a file, executes as code
- `input()` executes user's input as code!

Slide 18

PHP: Code Injection

- Intent: user views messages written to file

```
if ($_GET["action"] == "NewMessage") {
  /* save to file */
  echo "Message Saved!<p>\n";
 }
 else if ($_GET["action"] == "ViewMesgs") {
   include($MessageFile);
 }
```

- A URL-encoded message

```
message=%3C?php%20system(%22/bin/ls%20-l%22);?%3E
```

- will decode to

```
<?php system("/bin/ls -l");?>
```

and display list of files in directory!

Protection Against Command Injection

Avoid Calling Commands

- Commands are a user-level interface
- Programs should use library APIs
- Library functions for most filesystem commands exists (`chdir`, `mkdir`, `remove`, etc.)
- Cannot be manipulated into command execution

Escape Arguments

- When building a command string, escape arguments with OS-specific command characters / separators (`&` `;` `|` etc.)

- If possible, use libraries to do this, e.g. PHP: `escapeshellarg`, `escapeshellcmd`

Parameterize + Validate

- If possible, use structured APIs that separate data and commands

- Validate both commands and arguments

  - whitelisting with explicit allowed args
  - whitelist regular expressions (eliminate metacharacters like `$` `|` `&`)
  - encode metacharacters
  - quote arguments with spaces

More Defense Options

- Use sandboxing (limit privilege and scope of consequences)
- Reduce attack surface: limit data under external control (e.g., keep in session rather than send to client)
- Use environments that do automatic taint propagation (perl -T), forcing program to include steps that remove tainting.