Security Coding Module - Input Validation – “All Input is Evil” – CS1

Background

Summary:

Any program input – such as a user typing at a keyboard or a network connection – can potentially be the source of security vulnerabilities and disastrous bugs. All input should be treated as potentially dangerous.

Description:

Most software packages rely upon external input. Although information typed at a computer might be the most familiar, networks and external devices can also send data to a program. Generally, this data is of a specific type: for example, a user interface that requests a person’s name might be written to expect a series of alphabetic characters. If the correct type and form of data is provided, the program might work fine. However, if programs are not carefully written, attackers can construct inputs that can cause malicious code to be executed.

Risk – How Can It Happen?

Any data that can enter your program from an external source can be a potential source of problems. If external data is not checked to verify that it has the right type of information, the right amount of information, and the right structure of information, it can cause problems. Any program that processes data from external sources without adequate validation can be susceptible to security vulnerabilities.

Examples of Occurrence:
1. In December 2005, a Japanese securities trader made a $1 billion typing error, when he mistakenly sold 600,000 shares of stock at 1 yen each instead of selling one share for 600,000 yen. A few lines of code may have averted this error. Fat fingered typing costs a trader’s bosses £128m. The Times Online, December 09, 2005

2. Web applications are highly vulnerable to input validation errors. Inputting the invalid entry “!@#$%^&*()” on a vulnerable e-commerce site may cause performance issues or denial of service on a vulnerable system or invalid passwords such as “pwd” or “1=1—” may result in unauthorized access. http://www.processor.com/editorial/article.asp?article=articles%2Fp3112%2Fp32p32p32p32p32p32p12.asp&guid=&searchtype=&WordList=&bJumpTo=True

3. A Norwegian woman mistyped her account number on an internet banking system. Instead of typing her 11-digit account number, she accidentally typed an extra digit, for a total of 12 numbers. The system discarded the extra digit, and transferred $100,000 to the (incorrect) account. A simple dialog box informing her that she had typed too many digits would have helped avoid this expensive error. Olsen, Kai. "The $100,000 Keying error" IEEE Computer, August 2008

4. The site xssed.com lists nearly 13,000 vulnerable Web pages, including sites such as yahoo.com, google.com, msn.com, facebook.com, craigslist.com and cnn.com

5. The Risks digest (http://catless.ncl.ac.uk/Risks) – an invaluable resource on computing systems gone wrong – carried a report of an electronic commerce web site that failed to verify the quantity of items ordered. After accidentally typing “1.1” for the desired quantity of an item (instead of one), an amused customer found that the system would let him order 1.1 cocktail shakers at $9.99 each, for a total of $10.99. A simple check to verify that the quantity was an integer value would have eliminated the absurd possibility of ordering one-tenth of a cocktail shaker. Source: Richard Kaszeta, “Lack of sanity checking in Web shopping cart software “ Risks Digest, 23(51) http://catless.ncl.ac.uk/Risks/23.51.html#subj11

How Can I Properly Validate Input?

Check your input: Below is a partial list of some checks that you might want to include:

- **Range check** - numbers checked to ensure they are within a range of possible values, e.g., the value for month should lie between 1 and 12.
- **Reasonable check**: values are checked for their reasonableness, e.g. (age > 16) && (age < 100)
- **Divide by Zero**: variables are checked for values that might cause problems such as division by zero.
- **Length check**: variables are checked to ensure they are the appropriate length, for example, a US telephone number has 10 digits.
- **Format check** – Checks that the data is in a specified format (template), e.g., dates have to be in the format DD/MM/YYYY.
- **Available option** – Check items selected from a menu or sets of choices to ensure they are possible options: that is, always include a default when using a switch.

In many cases, this can be difficult. Checking a telephone number may require consideration of the many differing telephone formats used by countries around the world.

Recover Appropriately: A robust program will respond to invalid input in a manner that is appropriate, correct, and secure. When your program runs across invalid input, it should recover
as much as possible, and then repeat the request, or otherwise continue on. Arbitrary decisions such as truncating or otherwise reformatting data to “make it fit” should be avoided.

The following function shows input validation to check range and/or reasonableness:

```cpp
int ValidNum(int min, int max)
{
    int value;
    cin >> value;
    while ((value < min || value > max))
    {
        cout << "Enter a number between " << min << " and " << max << endl;
        cin >> value;
    }
    return value;
}
```

Laboratory Assignment

1. Write an input validation loop that asks the user to enter a body weight.
2. Write a program to calculate BMI = Weight (lbs)/Height (in)\(^2\) x 703
3. Complete the security checklist for this program. Submit marked program and completed checklist.
4. Add any additional input validation to your program that completing the checklist identified.

Security Checklist

<table>
<thead>
<tr>
<th>Security Checklist</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vulnerability:</strong> Failure to Validate Input <strong>Course:</strong> CS1</td>
</tr>
<tr>
<td>Check each line of code</td>
</tr>
<tr>
<td>1. Mark with a V each variable that is input</td>
</tr>
<tr>
<td>For each V, which of the following is applicable</td>
</tr>
<tr>
<td>1. Check length?</td>
</tr>
<tr>
<td>2. Check range (reasonableness)?</td>
</tr>
<tr>
<td>3. Check format?</td>
</tr>
</tbody>
</table>
Discussion Questions

1. Refer back to the lab on integer errors. Can improper input validation lead to integer errors?
2. Checking for division by zero is pretty straightforward. Checking for a possible integer overflow as a result of an operation can be difficult. Why?
3. Explain how range and reasonableness checking can help prevent integer overflow.
4. Why is a while loop usually a more effective way to perform input validation than an if..else?
5. Accepting known good values is known as whitelisting. Rejecting bad values is known as blacklisting. Write the loop construct for whitelisting a body temperature. Write the loop construct for blacklisting a body temperature. Why is whitelisting much stronger?
6. Explain the Simpson cartoon above. What is a “fat finger”?
7. List ways to make your program “fat finger friendly.”

Further Work (optional – check with your instructor if you need to answer the following questions)

1. Filenames are particularly vulnerable to security vulnerabilities. Research to find out why.
2. Another important security strategy is “defense in depth”. Explain what you think this means. How could this relate to input validation?