Secure Software Design

Andey Robins

Spring 23 - Week 11

Input Handling

Outline

- Untrusted Input
- Input Validation
- Vulnerabilities
- Mitigation
- Fuzz Testing

Untrusted Input

Untrusted Input are inputs out of your control (i.e. able to be manipulated by an attacker)

These are values you **should not trust**, in other words, they draw a trust boundary in their wake.

Input as a Threat Vector

What is the most wide open input vector in the world?

The Internet

"Doing a better job at input validation is perhaps the most impactful low-hanging fruit available to developrs for reducing vulnerabilities."

Input Validation

Several approaches:

1. Bounds checking
2. Specify encodings
3. Escaping

Determining Validity

Obviously, the first step in validating input is to define valid conditions.

"What usually works well is to establish an explicit limit on inputs and then leave plenty of headroom on the implementation to be certain of correctly processing all valid inputs."

Example: Guess-my-number

```
println!("Enter a number between 1-100")
let mut input_line = String::new();
io::stdin()
    .read_line(&mut input_line)?;
let guess: ??? = input_line.trim().parse()?;
```

Example: Guess-my-number

We can use the max size to make sure any number input is able to be stored.

```
println!("Enter a number between 1-100")
let mut input_line = String::new();
io::stdin()
    .read_line(&mut input_line)?;
let guess: i64 = input_line.trim().parse()?;
```

Example: Guess-my-number

But we only need to hold values from 1 to 100, so using the smallest possible size to indicate our intent may be better. This also includes some minor validation already.

```
println!("Enter a number between 1-100")
let mut input_line = String::new();
io::stdin().read_line(&mut input_line)?;
let guess: u8 = input_line.trim().parse()?;
```

Validation Criteria

Several overlaid criteria are checked to ensure the validity of input.

1. Use the definition of “validity” you put together in the last step
2. Check the size of your input
3. Check other common bounds

The XML “Zip Bomb”

4 GB in only a few lines of code.

```
<!DOCTYPE dtd[
    <!ENTITY big1 "big!">
    <!ENTITY big2 "&big1;&big1;&big1;&big1;&big1;
        &big1;&big1;&big1;">
    <!-- ... -->
    <!ENTITY big8 "&big7;&big7;&big7;&big7;&big7;
        &big7;&big7;&big7;">
]>
<mega>&big8;&big8;&big8;&big8;&big8;&big8;
    &big8;&big8;</mega>
```

Common Criteria: Numbers

- range of datatype
- precision of floats
- size

Common Criteria: Strings

- encoding schema
- length
- special characters
- potentially syntax

Aside: Bytes vs Characters

"I recommend specifying maximum string lengths in characters rather than bytes, if only so that non-programmers have some hope of knowing what this constraint means." - Kohnfelder

"I recommend specifying maximum string lengths in bytes rather than characters, otherwise, you're waiting for UTF-8/16 to come wreck your application." - Andey

Rejection of Invalid Input

What is the rule for swimming in the ocean?

When in doubt, don't go out.

Rejection of Invalid Input

Applying that rule, when in doubt, reject the input.

Complete acceptance and rejection is the cleanest and easiest to reason about.

Rejection Best Practices

1. Explain what constitutes a valid entry
2. Flag as many errors at once as possible
3. Have clear, simple rules for human input
4. Break up extended input into parts

Correcting Invalid Input

Humans are not computer [citation needed]. So strict input rules are not always the easiest for us to handle.

Attempting to correct erroneous inputs is a great way to increase the availability of your system.

Example: Phone Number Input

```
function adddashes() {
  f = document.getElementById('phone');
  f.value = f.value.slice(0, 3) + "-" +
            f.value.slice(3, 6) + "-" +
            f.value.slice(6, 10);
}
```

Best Practices

1. Remove syntactic requirements to make input easier
2. Present “best gueses” for the user to select from
3. Format their input as they provide it

Validation Challenges

1. How do we validate a phone number input of 11 digits?
2. How do we validate a username with Katakana?
3. How do we validate an email is valid during signup?

Input Vulnerabilities

A number of potential places are rife with vulnerabilities caused by injection:

- Buffers
- Glyphs/encodings
- Case changes

Length Problems

“Don’t confuse character count with byte length when allocating buffers.”

Full stop, that’s the advice. Because. . .

Unicode

- Most things use UTF-8; Also there are UTF-7, UTF-16, and UTF-32
- Sorting unicode behaves different across languages and code points
- A “character” is a variable number of bytes
- Modifiers, diachritics, and symbols are in there too
- Localization can change Unicode valuation

How to Handle Unicode

Dont. At all costs.

Use a library, don't reinvent the wheel. Most languages have a strong Unicode handler either as part of the Standard library or as a widely recognized community package.

Encodings

Unicode encodes characters, not glyphs

Case Change Vulnerabilities

Lowercase characters to uppercase characters is a many-to-one relationship.

Injection Vulnerabilities

Injection Attacks are attacks which use an input to a system to be directly interpreted or executed in a malicious way.

Example: My intramural team named “No Game Scheduled.”

Common Injection Targets

- Filepaths
- Shell commands
- HTTP
- **SQL**

SQL Injection

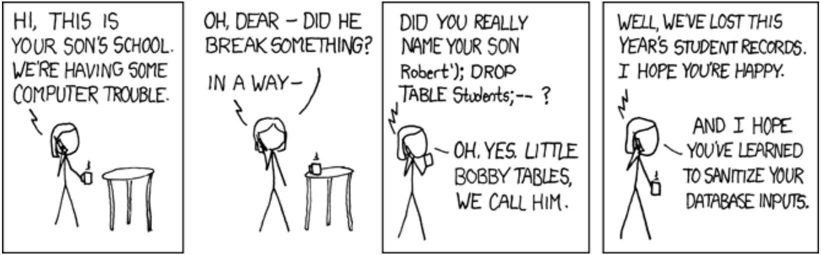
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 1: Bobby Tables

SQL Injection Example

```
INSERT INTO Students (name)
VALUES ('Robert');
```

Where the input string is `Robert`

```sql
INSERT INTO Students (name)
VALUES ('Robert'); DROP TABLE Students;--)
```

Where the input string is `Robert'); DROP TABLE Students;--`

Underlying Cause

```
let name = 'Robert';
let injection = "Robert'); DROP TABLE Students; --";
// name = injection;
let sql_stmt = `
    INSERT INTO Students (name)
    VALUES (${name});
`;
```

Safe Input

```
// both are fine
db.run(`
    INSERT INTO Students (name)
    VALUES (:name);
`, {
    ":name": name,
});
db.run(`
    INSERT INTO Students (name)
    VALUES (:name);
`, {
    ":name": injection,
});
```

Regular Expressions

Our best tool for validating strings, they're a state-machine based parser.

Can be an exploit point for DoS attacks where backtracking is possible.

Regex Injection

```python
import re
from time import time

def time_backtrack(n=1):
    start = time()
    re.match(r'(D+)+$', 'D'*n + '!')
    return time() - start

for i in range(1, 40):
    print(i, ':=', time_backtrack(i))
```

Mitigation

1. Anytime you’re concatenating strings and vars, consider injection attacks
2. Don’t insert untrusted data into any strings
3. Make use of helper functions for your domain

Fuzzing

Fuzzing is an automated testing technique which makes attempts a wide range of random potential inputs to ensure the system behaves well in general with any arbitrary input.

Code reviewed externally

Questions?

Next Time

Welcome to the Internet.