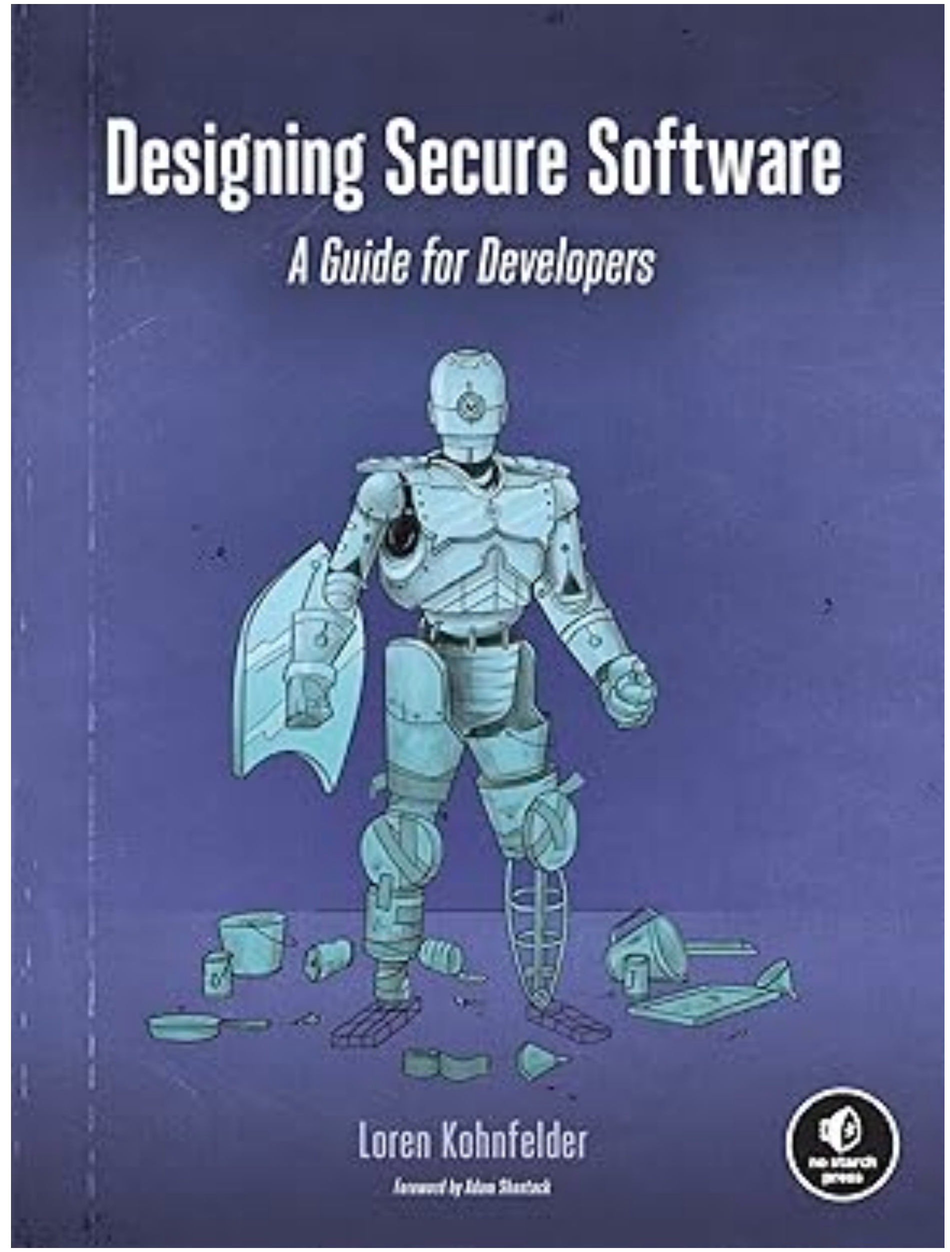


10 Untrusted Input

Updated 4-20-24

Topics

- Input Validation
  - Determining Validity
  - Validation Criteria
  - Rejecting Invalid Input
  - Correcting Invalid Input
- Character String Vulnerabilities
  - Length Issues
  - Unicode Issues

Topics

- Injection Vulnerabilities
  - SQL Injection
  - Path Traversal
  - Regular Expressions
  - Dangers of XML
- Mitigating Injection Attacks

Input Validation

Input Validation Examples

- When logging in
  - Ensure that username contains only 8-40 valid characters
- Accepting a number of hours for a week's pay
  - Limit it to 100 max
- **Attack Surface**
  - Obviously inputs from the Internet are untrusted
  - Or from users
  - But inputs from other modules of code may be harmful too
  - Because of changes as code is updated

Determining Validity

- Must anticipate all future valid input values
  - And disallow the rest
- Allow some headroom
  - Allocate a 4096-byte buffer
  - Limit inputs to 4000 bytes

Validation Criteria

- Input must
  - Not exceed maximum size
  - Be in proper format
  - Be within a range of acceptable values
- Size limit prevents DoS attacks caused by inputting large amounts of data
- Formats include digits, strings with certain allowed characters. XML, JSON
- Do the three tests in the order shown above

Understandable Limits

- Make your limits understandable to non-programmers
- 100 characters, not 100 bytes
- 1,000,000 products, not $2^{32} - 1 = 4,292,967,295$

Rejecting Invalid Input

- Safest approach
- If input comes from a user
- It's kind to provide an informative error message
  - To help the user provide valid input

Best Practices

- Explain what constitutes a valid entry as part of the user interface, saving at least those who read it from having to guess and retry. (How am I supposed to know that area codes should be hyphenated rather than parenthesized?)
- Flag multiple errors at once, so they can be corrected and resubmitted in one step.
- When people are directly providing the input, keep the rules simple and clear.
- Break up complicated forms into parts, with a separate form for each part, so people can see that they're making progress.

Rejecting Inputs from Other Computers

- Write documentation precisely describing the constraints
- Fully rejecting input is safer than trying to clean it and use it
  - The error indicates that something is wrong, so it can be fixed

Correcting Invalid Input

- You may not want to stop the process for a minor error
  - Lost sales, frustrated customers...
- Attempt to correct invalid input
  - Truncate long strings
  - Remove leading or trailing spaces
- Correcting addresses is complicated
- May change input in an unintended fashion
  - Such as stripping country codes off long phone numbers

Character String Vulnerabilities

Length Issues

- Long strings may cause buffer overflows
  - Or performance problems if they are very long
- So limit maximum number of characters

Unicode Issues

- UTF-8 is most common encoding
- One character can be 1-4 bytes long

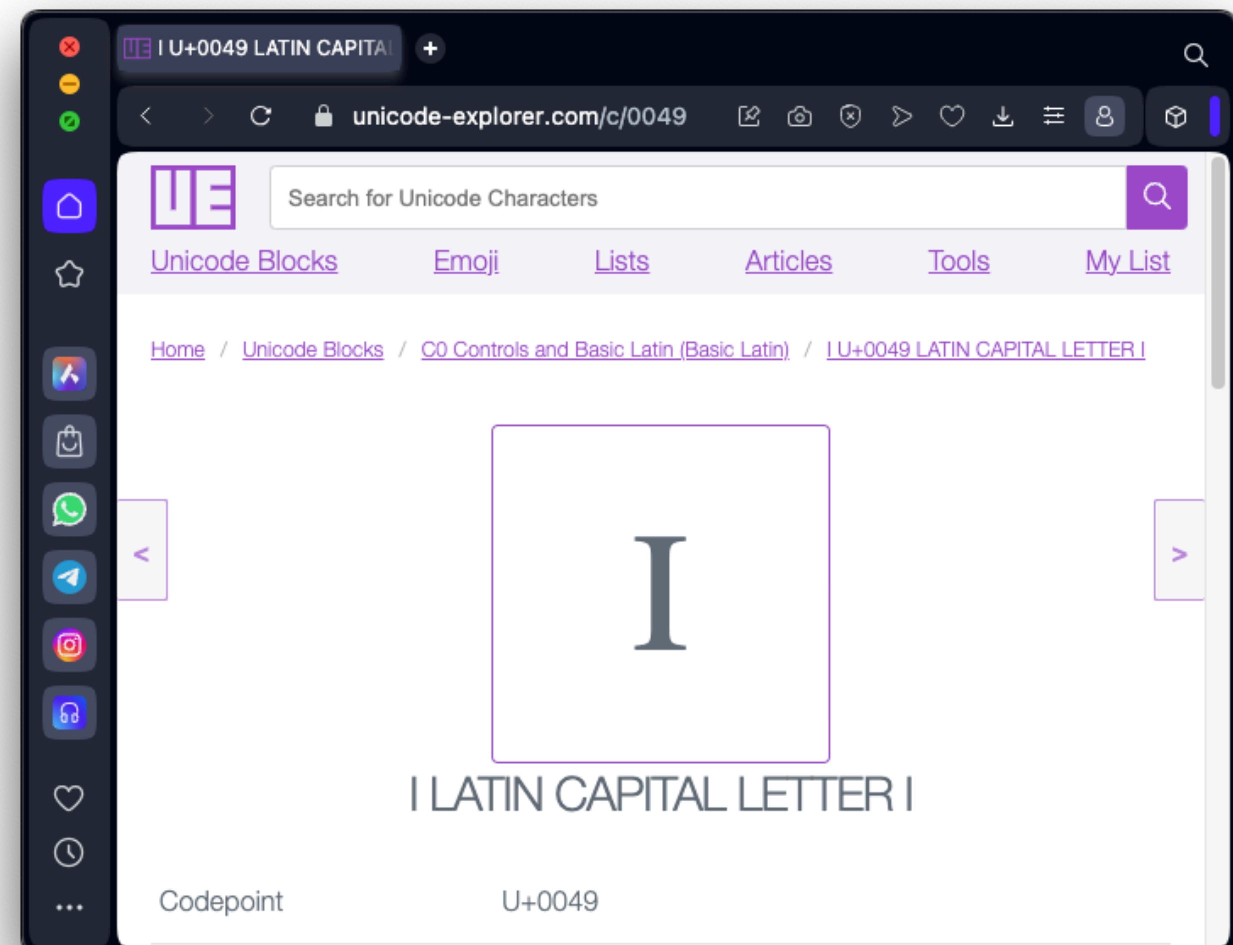
Code point ↔ UTF-8 conversion

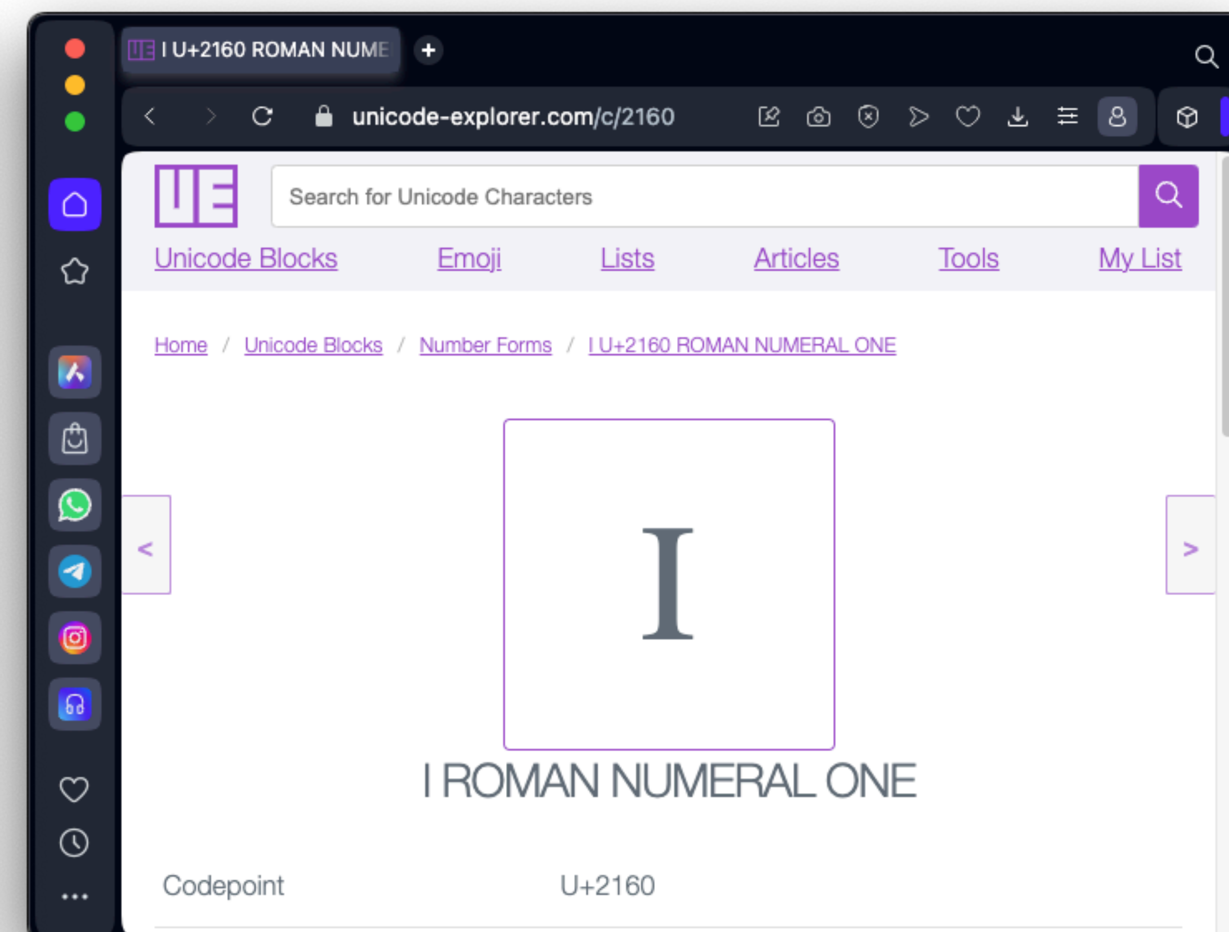
First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- There are also UTF=7, UTF-16, and UTF-32 encodings

Encodings and Glyphs

- Glyphs are the rendered visual forms of characters
- These two characters are different but have the same glyphs





Homomorphs

- Different characters with identical glyphs
- Often used by attackers to fool users
- Spelling Paypal with a Cyrillic character U+0420 instead of P
- The Latin letter Ç (U+00C7) also has a two-character representation, consisting of a capital C (U+0043) followed by the “Combining Cedilla” character (U+0327).

Canonicalization

- A common coding strategy
- Normalizing input strings to a standard form
- Not simple for Unicode

Case Change

- Converting all characters to lowercase or UPPERCASE
  - Simplifies later processing
- But some characters have surprising properties
- '**This ıs a test.**' and '**This is a test.**'
- Converted to uppercase, they both turn into '**THIS IS A TEST.**'
- Lowercase dotless ı (U+0131) and
  - The familiar lowercase i (U+0069)
- Both become uppercase I (U+0049).

Blocking <script>

- Filtering algorithm:
    - Convert input to lowercase
    - Scan for **<script>**
    - Convert to uppercase for output
- **<scrıpt>** will pass this test

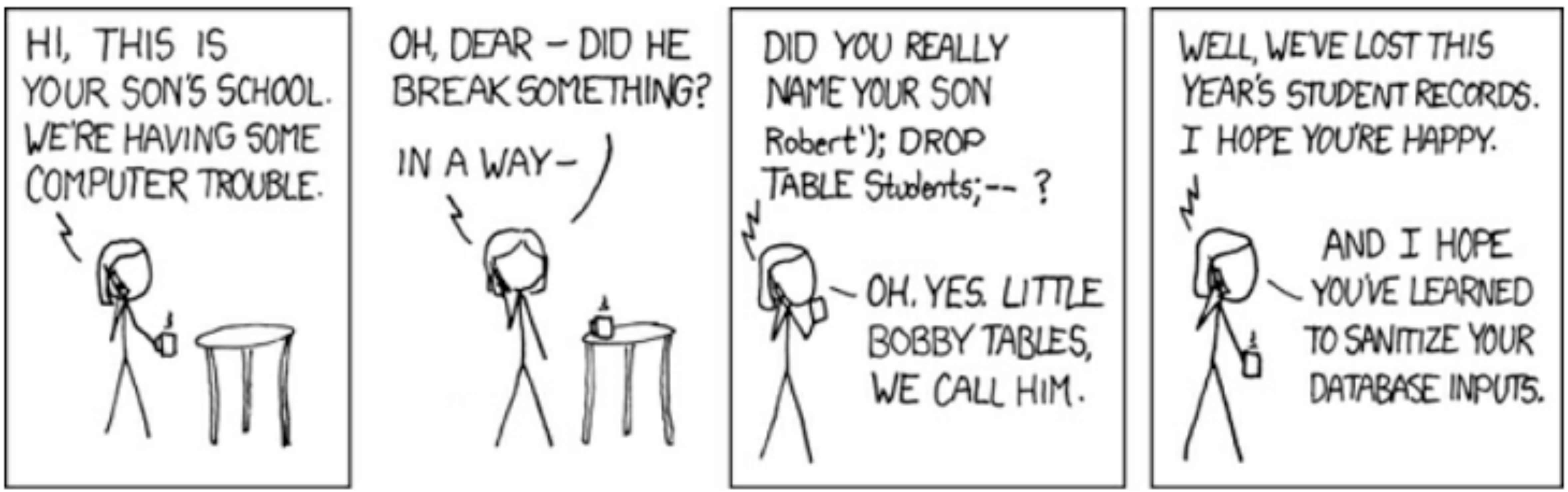
Injection Vulnerabilities

Forms of Injection

- Data from the user is interpreted as commands at the server

  - **SQL statements**
  - **Filepath names**
  - **Regular expressions (as a DoS threat)**
  - **XML data (specifically, XXE declarations)**
  - **Shell commands**
  - **Interpreting strings as code (for example, JavaScript's `eval` function)**
  - **HTML and HTTP headers (covered in Chapter 11)**

SQL Injection



- Exploits of a Mom

How it Works

- Normal student name: **Robert**

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

- Malicious student name

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

- What the server sees

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

Vulnerable Code

```
sql_stmt = "INSERT INTO Students (name) VALUES ('" + student_name +
"');";
```

- Includes input without validating it first
- Simple defense: block apostrophes in names
- BUT some names contain apostrophes

Least Privilege

- Software registering students should not have administrative privileges
  - Ability to delete tables

Vulnerable Code

```
import sqlite3
con = sqlite3.connect('school.db')
student_name = "Robert'); DROP TABLE Students;--"
# The WRONG way to query the database follows:
sql_stmt = "INSERT INTO Students (name) VALUES ('" + student_name +
"');"
con.executescript(sql_stmt)
```

Fixed Code

```
import sqlite3
con = sqlite3.connect('school.db')
student_name = "Robert'); DROP TABLE Students;--"
# The RIGHT way to query the database follows:
con.execute("INSERT INTO Students (name) VALUES (?)", (student_name,))
```

- **(?)** place holder is filled in from the **student_name** value
- No apostrophes used
- No chance of misinterpreting the name as executable code

Changing a Grade

- This attack doesn't require a second SQL statement
- Student name: **Robert', 'A+'); --**
- When submitting grades:

```
INSERT INTO Grades (name, grade) VALUES ('Robert', 'F');
```

But with the name `Robert', 'A+');--` **that command becomes:**

```
INSERT INTO Grades (name, grade) VALUES ('Robert', 'A+');--', 'F');
```

Path Traversal

- Input is a filename **`x`**
- Used to fetch an image from **`/server/data/image_store/x`**
- Attack: set **`x`** to **`../../secret/key`**
- These are equivalent path names:
    - `/server/data/image_store/../../secret/key`
    - `/server/data/../secret/key`
    - `/server/secret/key`

Defense

- Ensure that input contains only alphanumeric characters
- Or filter out troublesome characters like **..** and **/**
- BUT Windows uses ****

Vulnerable Algorithm

- If path begins with **../**, reject it
- BUT an attacker who knows the name of a subfolder can use
  - **subfolder/../../../secret/key**

Fixed Code

```
SC — nano safe_path_demo.py — 73×23
  UW PICO 5.09                     File: safe_path_demo.py

import os

def safe_path(path):
   """Checks that argument path is a safe file path.
   If not, returns None. If safe, returns the
   normalized absolute file path. """

   base_dir = os.path.dirname(os.path.abspath(__file__))
              # __file__ is the path to this running script

   filepath = os.path.normpath(os.path.join(base_dir, path))
   if base_dir != os.path.commonpath([base_dir, filepath]):
     return None
   return filepath

path = input("Path:")
print( safe_path(path) )


^G Get Help ^O WriteOut ^R Read File^Y Prev Pg  ^K Cut Text ^C Cur Pos
^X Exit     ^J Justify  ^W Where is ^V Next Pg  ^U UnCut Tex^T To Spell
```

Demo

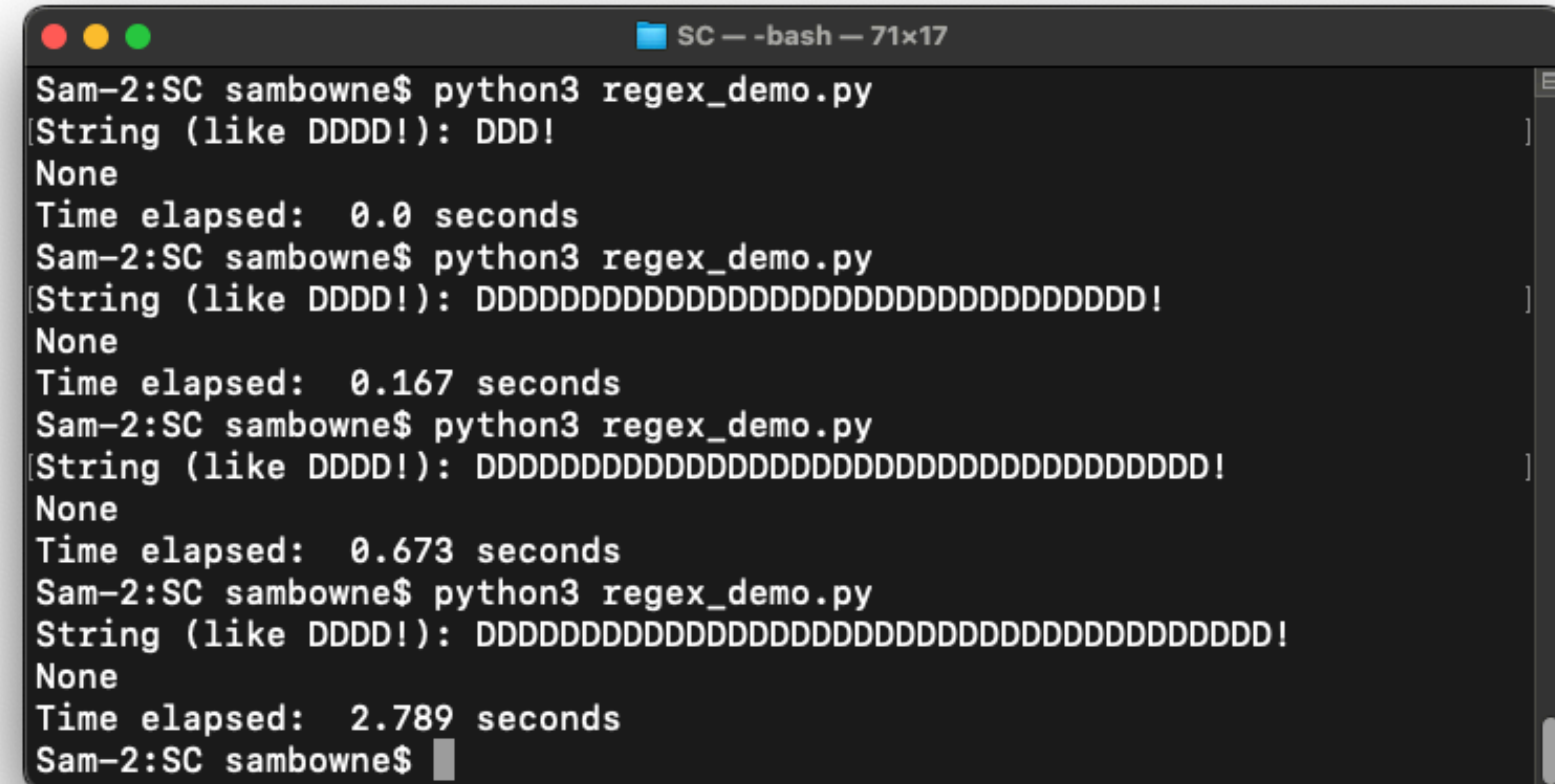
```
Sam-2:SC sambowne$ python3 safe_path_demo.py
Path:foo
/Users/sambowne/Downloads/SC/foo
Sam-2:SC sambowne$ python3 safe_path_demo.py
Path:../../foo
None
Sam-2:SC sambowne$ python3 safe_path_demo.py
Path:subfolder/../../../foo
None
Sam-2:SC sambowne$
```

Regular Expressions

- Some expressions require backtracking and take a long time

```
SC — nano regex_demo.py — 68×17
  UW PICO 5.09              File: regex_demo.py              Modified

import re, time

str = input("String (like DDDD!): ")
t0 = time.time()

print(re.match(r'(DD+)+$', str))

t = time.time() - t0
print( "Time elapsed: ", round(t,3), "seconds" )



^G Get Help^O WriteOut^R Read Fil^Y Prev Pg ^K Cut Text^C Cur Pos
^X Exit    ^J Justify ^W Where is^V Next Pg ^U UnCut Te^T To Spell
```

Time Required



Mitigation

- Avoid letting untrusted inputs influence computations that have the potential to blow up
- Don't let untrusted inputs define the regex
- Limit the length of the string the regex matches
- Test the worst-case to ensure it's not too slow

Dangers of XML

- XML entity declarations
- This code generates 8 megabytes of XML

```
<!DOCTYPE dtd[ <!ENTITY big1 "big!"> <!ENTITY big2
"&big1;&big1;&big1;&big1;&big1;&big1;&big1;&big1;"> <!ENTITY big3
"&big2;&big2;&big2;&big2;&big2;&big2;&big2;&big2;"> <!ENTITY big4
"&big3;&big3;&big3;&big3;&big3;&big3;&big3;&big3;"> <!ENTITY big5
"&big4;&big4;&big4;&big4;&big4;&big4;&big4;&big4;"> <!ENTITY big6
"&big5;&big5;&big5;&big5;&big5;&big5;&big5;&big5;"> <!ENTITY big7
"&big6;&big6;&big6;&big6;&big6;&big6;&big6;&big6;">
]>
<mega>&big7;&big7;&big7;&big7;&big7;&big7;&big7;&big7;</mega>
```

Reading a File

- This code puts the contents of the passwd file into &snoop;

```
<!ENTITY snoop SYSTEM "file:///etc/passwd>" >
```

- **Defense**
  - Keep untrusted inputs out of any XML your code uses

Mitigating Injection Attacks

- Input validation is the first line of defense
  - But may not be enough
- Avoid inserting untrusted data into constructed strings for execution
- Use trusted libraries with safe ways to use data in SQL
- Use direct system call to **readdir(3)** instead of constructing a command starting with **ls**
  - Cannot execute any other command
- Avoid storing data in the filesystem directly
  - Anticipating and blocking all possible attacks is tricky

Source Code Scanners

- Easily find insecure SQL, exec, eval, etc.

Kahoot!

Ch 10