



Secure Software Design and Engineering (CY-321)

Input Validation

Dr. Zubair Ahmad

A kind Reminder

- Attendance?
 - Active Attendance
 - **Dead Bodies.**
 - **Active Minds**
 - Mobiles in hands -> Mark as absent
 - 80% mandatory

Trusting Input

Trust in input is often not warranted and sometimes downright dangerous.

```
#include <stdio.h>

int main() {
    int a;
    scanf("%d", &a);
    printf("%d\n", a);
    return 0;
}
```

What happens if
the user enters
something that
is not a
number?

The value of a is undefined, and therefore could be anything.

Trusting Input

```
#include <stdio.h>
#include <string.h>

int main() {
    char filename[1024];
    char command[sizeof(filename) + 4];

    fgets(filename, sizeof(filename), stdin);
    filename[sizeof(filename) - 1] = '\\0';

    strcpy(command, "cat ");
    strcat(command, filename);

    system(command); /* Executes a shell */

    return 0;
}
```

That is more interesting, Is there a buffer overflow? **No.**

What other problems might there be?

. What happens if a user enters **“/dev/null;
rm -rf *”**?

Trusting Input *(SQL Injection)*

```
import sqlite3

def n_rows(table, db_connection):
    """Return number of rows in
    TABLE."""
    query_start = "SELECT COUNT(*)
    FROM "
    query = query_start + table

    cursor = db_connection.cursor()
    cursor.execute(query)
    result = cursor.fetchone()

    return result[0] if result else 0
```

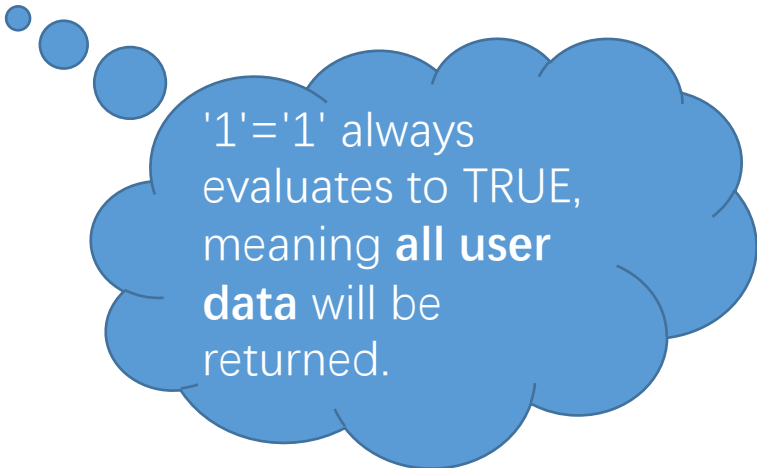
What if the argument isn't checked and the user can somehow enter **"customers; DROP TABLE customers"** ?

Trusting Input *(SQL Injection)*

```
import sqlite3

def get_user_data(username, db_connection):
    """Retrieve user data based on the
    username."""
    query = f"SELECT * FROM users WHERE
    username = '{username}'"

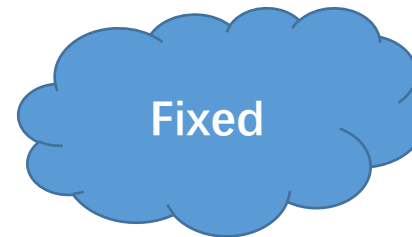
    cursor = db_connection.cursor()
    cursor.execute(query)
    return cursor.fetchall()
```



'1'='1' always evaluates to TRUE, meaning **all user data** will be returned.

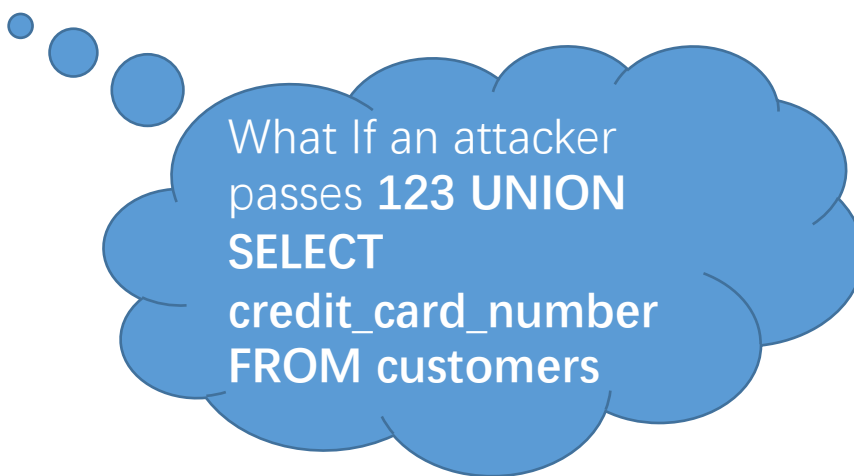
Trusting Input

```
def get_user_data(username, db_connection):  
    """Retrieve user data safely."""  
    query = "SELECT * FROM users WHERE  
username = ?"  
  
    cursor = db_connection.cursor()  
    cursor.execute(query, (username,))  
    return cursor.fetchall()
```



Trusting Input *(SQL Injection)*

```
def get_balance(account_id, db_connection):  
    """Retrieve account balance."""  
    query = f"SELECT balance FROM accounts  
WHERE account_id = {account_id}"  
  
    cursor = db_connection.cursor()  
    cursor.execute(query)  
    return cursor.fetchall()
```



What If an attacker
passes **123 UNION
SELECT
credit_card_number
FROM customers**

Trusting Input

```
def get_balance(account_id,  
db_connection):  
    """Retrieve account balance safely."""  
    query = "SELECT balance FROM accounts  
WHERE account_id = ?"  
  
    cursor = db_connection.cursor()  
    cursor.execute(query, (account_id,))  
    return cursor.fetchall()
```



Fixed

Trusting Input *(OS Command Injection)*

```
import os

def delete_file(filename):
    """Deletes a file given its name."""
    command = f"rm -rf {filename}"
    os.system(command)
```

```
# Example usage
delete_file("important.txt")
```

```
delete_file("important.txt; rm -rf /")
```

```
rm -rf important.txt; rm -rf /
```



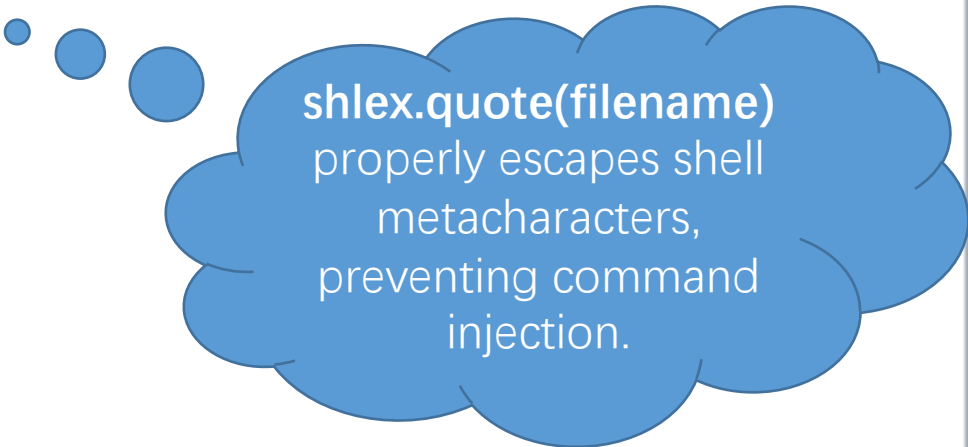
How an
Attacker
Exploits This

Trusting Input

```
import os
import shlex

def delete_file(filename):
    """Safely deletes a file by using
    shlex.quote()."""
    safe_filename =
shlex.quote(filename) os.system(f"rm -
rf {safe_filename}")

# Example usage
delete_file("important.txt")
```



shlex.quote(filename)
properly escapes shell
metacharacters,
preventing command
injection.

Trusting Input *(OS Command Injection)*

```
import subprocess

def ping_host(host):
    """Pings a given host."""
    command = f"ping -c 4 {host}"
    subprocess.Popen(command, shell=True)
```

```
# Example usage
ping_host("example.com")
```

```
ping_host("example.com && cat /etc/passwd")
```

```
ping -c 4 example.com && cat /etc/passwd
```



How an
Attacker
Exploits This

Trusting Input

```
import subprocess

def ping_host(host):
    """Safely pings a host using
    subprocess.run()."""
    subprocess.run(["ping", "-c", "4", host],
check=True)

# Example usage
ping_host("example.com")
```

Uses a **list of arguments** instead of **shell=True**, preventing command injection

Trusting Input (*LDAP Injection*)

```
import ldap

username = input("Enter username: ")
password = input("Enter password: ")

# LDAP connection
conn =
ldap.initialize("ldap://example.com")
conn.simple_bind_s("cn=admin,dc=example,dc=com", "adminpassword")

search_filter = f"(uid={username})"
result =
conn.search_s("dc=example,dc=com",
ldap.SCOPE_SUBTREE, search_filter)

if result:
    print("Authentication successful")
else:
    print("Authentication failed")
```

Use LDAP escaping functions
(`ldap.filter.escape_filter_chars()`)

Restrict permissions in LDAP

Implement role-based access control
(RBAC)

Trusting Input (*XML Injection*)

Part-1

```
from lxml import etree

# XML Database
xml_data = """
<users>
  <user>
    <username>admin</username>
    <password>supersecret</password>
    <email>admin@example.com</email>
  </user>
  <user>
    <username>john</username>
    <password>mypassword</password>
    <email>john@example.com</email>
  </user>
</users>
"""
```

Part-2

```
# Load XML Data
tree = etree.XML(xml_data)

# Attacker input
username = input("Enter username: ")

# Vulnerable XQuery-like search
query = f"//user[username='{username}']/email"
result = tree.xpath(query)

if result:
    print("Extracted Email:", result[0].text)
else:
    print("User not found")
```

Trusting Input (*XML Injection*)

```
query = f"//user[username='{username}']/email"
```

With Attacker Input

```
query = "//user[username='admin' or contains(email, 'example.com')]/email"
```


Input Validation is Trust Management

A trust relationship is a relationship among the different participants in a software system and concerns the assumptions that those participants make about security properties of the other part.

For example, a function might assume that its inputs are shorter than some maximum length; or it might assume that its input is a valid user name

Why is Trust Management So Difficult?

Often, programmers extend trust to other parts of a program without realizing it.

Assumptions are easy to make: Object-Oriented programming has taught us that we must decompose a system into small, largely independent objects and that it is OK to forget about the big picture when we're coding individual objects.

Therefore, programmers are encouraged to think about software development in small steps.

Why is Trust Management So Difficult?

It is often easier to put the burden of validation on the **caller** instead of validating input in the **callee** because there is often no standard way to signal an error to the caller:

If input validation is done in the **caller**, the function being called (**callee**) can assume it always receives valid input.

If the **callee** is responsible for validation and finds an error, it needs a way to signal the error back to the **caller**.

Why is Trust Management So Difficult?



Ken Thompson invented Unix together with Dennis Richie.

For this achievement, he was awarded the ACM Turing Award in 1984 (a highly appropriate year).

In his award lecture, he outlined how he modified the Unix C compiler so that he got access to any Unix system.

Reflections on Trusting Trust

He modified the system such that the compiler source code was free of any trace of malicious activity.

If the C compiler detected that the login program was compiled, it compiled in a back door that would allow Thompson access with a special user name/password combination

Now, this modification is pretty obvious, because you can see it in the C compiler source code.

Reflections on Trusting Trust

1. He compiled the C compiler with itself;
2. He removed the modifications from the C compiler;
3. He recompiled the C compiler with itself one more time.

That way, all traces in the source code were gone and literally no amount of source code analysis would find any problems with the compiler.

Reflections on Trusting Trust

“The moral is obvious. You cant trust code that you did not totally create yourself. (Especially code from companies that employ people like me.)”

—Ken Thompson

Some Hard-And-Fast Rules

Consider all input to be untrusted and validate all user input

Encode output using the appropriate character set, escape special characters and quote input

Use structured mechanisms to separate data from code

Display generic error messages that yield minimal to no additional information.

Some Hard-And-Fast Rules

Implement fail safe by redirecting all errors to a generic error page and logging it for later review

Remove any unused functions or procedures from the database server if not needed

To mitigate OS command injection, run the code in a **sandbox** environment that enforces strict boundaries between the processes being executed and the Operating System

Some Hard-And-Fast Rules

Audit and log the queries that are executed along with their response times to detect injection attacks

Implement least privilege by using views, and restricting tables, queries and procedures to only the authorized set of users and/or accounts.

Use runtime policy enforcement to create the list of allowable commands (whitelist) and reject any command that does not match the whitelist.

Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby