اونيۏرسيتي مليسيا ڤهڠ
UNIVERSITI MALAYSIA PAHANG
FAKULTI KOMPUTERAN

Data & Network Security

Chapter 8 – Defensive Programming

Outline

Introduction

- Defensive Programming is a technique where assume the worst from all input.
- Also known as Proactive Debugging
  - Like defensive driving
  - Main goal – protect yourself from bad data or other factors you cannot control
- The first rule of defensive programming is :
  - Never Assume
  - Must always validate assumptions
  - Needs an awareness of the consequences of failures and the techniques used by attackers

Input validation

- Data from the user cannot be trusted.
- As such, all input must be validated.
- For each input:
    - Define the set of all legal input values.
    - When receiving input, validate against set of rules.
    - Determine the behaviour when input is incorrect:
        - Terminate
        - Retry
        - Warning

Input validation-Validation Example

- Lets assume input expect a monetary value.
  - Is the amount numeric?
  - Is the amount large enough or small enough?
  - Is it positive?
  - What decimal symbol was used?
  - How many decimal point does it have? (ex: 20.2555$)
  - Is it only composed of number? (ex: 10+25 is considered numeric by some systems)
  - The input length matching

Data sanitization

- Data sanitization - the removal of malicious data from user input, such as form submissions
- Avoid code-conflicts (duplicate ids for instance), security issues (xss codes etc), or other issues that might arise from non-standardized input & human error/deviance.
- Removing vulgarities & odd symbols from text to removing SQL injection attempts and other malicious code intrusion attempts.

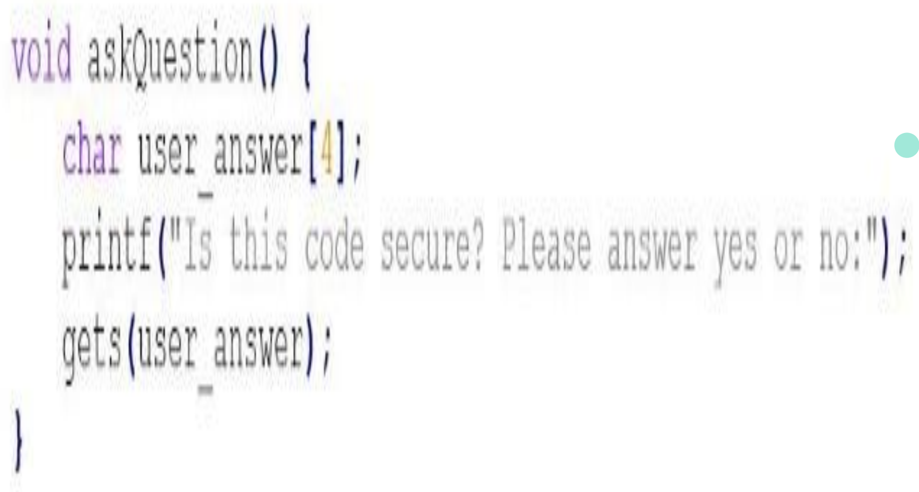
Examples of input validation and data sanitization errors:

- Buffer overflows
- Integer errors
- SQL injection
- XSS vulnerability

Buffer Overflows (Kerestan, 2017)

- Buffer overflow vulnerabilities deal with buffers, or memory allocations in languages that offer direct, low-level access to read and write memory.
- The case of languages such as C and Assembly, reading from or writing to one of these allocations does not entail any automatic bounds checking.
- The program can “overflow” the capacity of the buffer. This results in data being written past its end and overwriting the contents of subsequent addresses on the stack or heap, or extra data being read

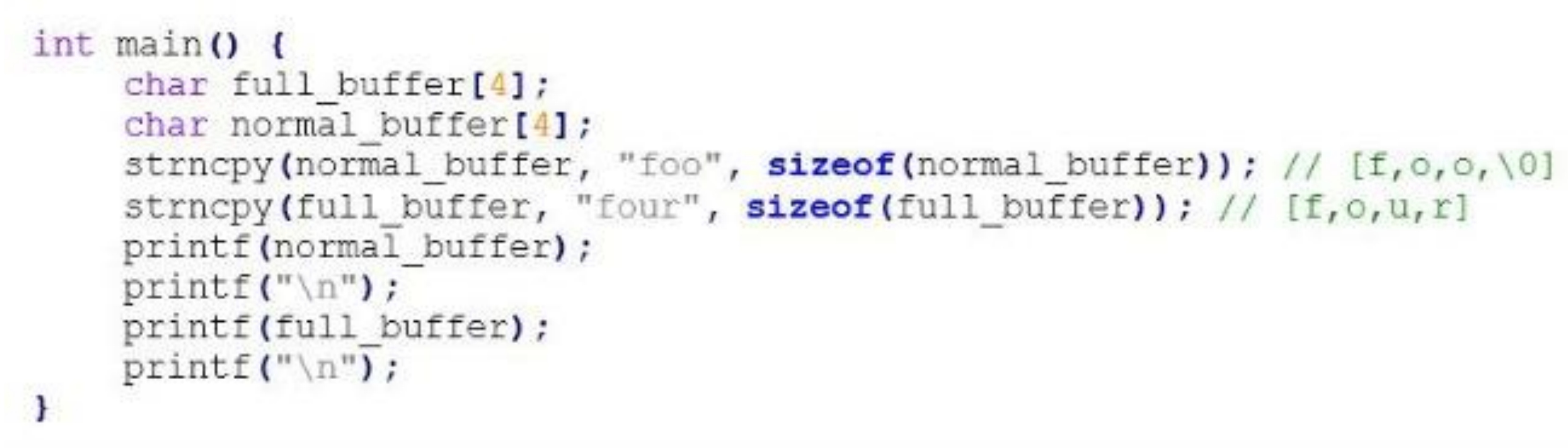
Buffer Overflows - Continue



```
void askQuestion() {
    char user_answer[4];
    printf("Is this code secure? Please answer yes or no:");
    gets(user_answer);
}
```

- The code is clearly shows that no bounds checking is performed.
- If the user enters “maybe” then the program will likely crash rather than asking the user for a valid answer and re-prompting with the question.
- The user’s answer is simply written into the buffer, regardless of its length.

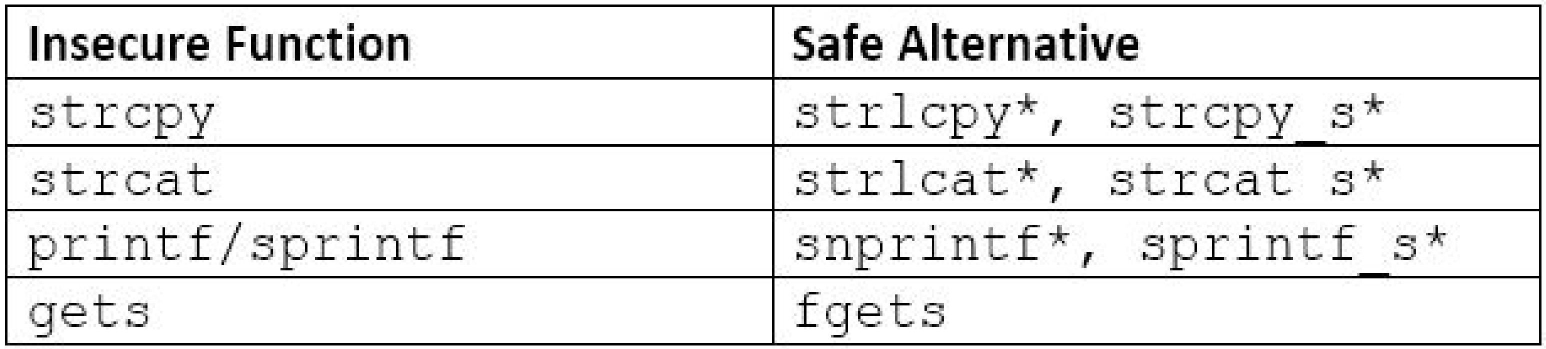
Buffer Overflows - Continue



```
int main() {
    char full_buffer[4];
    char normal_buffer[4];
    strncpy(normal_buffer, "foo", sizeof(normal_buffer)); // [f,o,o,\0]
    strncpy(full_buffer, "four", sizeof(full_buffer)); // [f,o,u,r]
    printf(normal_buffer);
    printf("\n");
    printf(full_buffer);
    printf("\n");
}
```

- The easiest way to prevent these vulnerabilities is to simply use a language that does not allow for them
  - C allows these vulnerabilities
  - Java, Python, and .NET, among other languages and platforms, don't require special checks
- Above code can cause over print for full_buffer. This happened because of null buffer at normal_buffer .

Buffer Overflows - Continue



Insecure Function	Safe Alternative
strcpy	strlcpy*, strcpy_s*
strcat	strlcat*, strcat_s*
printf/sprintf	snprintf*, sprintf_s*
gets	fgets

- The use of the secure alternatives listed above are preferable. When that is not possible, it is necessary to perform manual bounds checking and null termination when handling string buffers.

Integer errors

- Integer overflow occurs when you try to express a number that is larger than the largest number the integer type can handle.
- Example: user express the number 300 in one byte, you have an integer overflow (maximum is 255). 100,000 in two bytes is also an integer overflow (65,535 is the maximum).

Integer Error - Example

```
if ((y > 0 && x > INT_MAX - y) || (y < 0 && x <
INT_MIN - y))
{
        printf(“Integer Overflow”);
}
else {
        sum = x + y;
}
```

SQL injection

- injection attack wherein an attacker can execute malicious SQL statements (also commonly referred to as a malicious *payload*) that control a web application’s database server
- SQL Injection vulnerability is the oldest, most prevalent and most dangerous of web application vulnerabilities.

Example

SQL

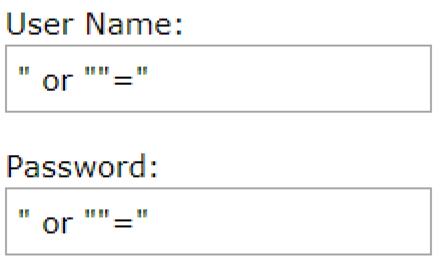
```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'
```

Result

```
SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
```

Let say Hacker input as below



User Name:

" or ""="

Password:

" or ""="

Result

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

SQL engine will generate as above that or is always TRUE

SQL Injection Protection

- use SQL parameters
- values that are added to an SQL query at execution time
- parameters are represented in the SQL statement by a @ marker
- SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed

Example

ASP .NET

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
db.Execute(txtSQL,txtUserId);
```

PHP

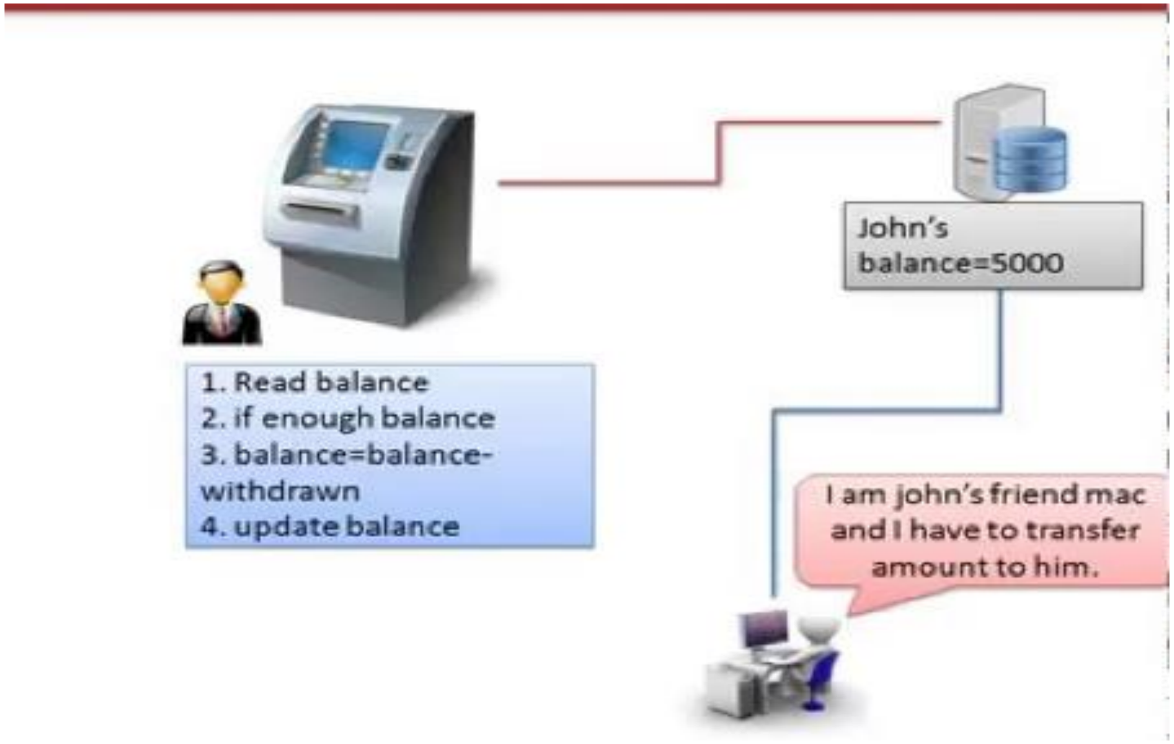
```
$stmt = $dbh->prepare("INSERT INTO Customers (CustomerName,Address,City)
VALUES (:nam, :add, :cit)");
$stmt->bindParam(':nam', $txtNam);
$stmt->bindParam(':add', $txtAdd);
$stmt->bindParam(':cit', $txtCit);
$stmt->execute();
```

XSS vulnerability

- Cross-Site Scripting (XSS) - web-based attack performed on vulnerable web applications.
- In XSS attacks, the victim is the user and not the application.
- In XSS attacks, malicious content is delivered to users using JavaScript.
- XSS - can have consequences such as tampering and sensitive data theft
- Attack XSS:
    - Hijack an account.
    - Spread web worms.
    - Access browser history and clipboard contents.
    - Control the browser remotely.
    - Scan and exploit intranet appliances and applications.

Race Condition

- Race Condition occurs when a second thread modifies the state of one


John's balance=5000
1. Read balance
2. if enough balance
3. balance=balance-withdrawn
4. update balance
I am john's friend mac and I have to transfer amount to him.

Race Condition - Prevention

- make sure that the critical section is executed as an atomic instruction.
    - That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.
- using proper thread synchronization in critical sections.
    - using a **synchronized block of Java code**.
    - other synchronization constructs like **locks** or atomic variable like **java.util.concurrent.atomic.AtomicInteger**.

Correct usage of third-party components

- Have clear policy
- monitor security patches
- Know that components used may not rely on other components that may not be secure.
- Frequently check for third library used.

Choice of programming language and type-safe languages

- Characteristic for selecting the programming language
  - Ease of learning
  - Ease of understanding
  - Speed of development
  - Help with enforcement of correct code
  - Performance of compiled code
  - Supported platform environments
  - Portability
  - Fit-for-purpose

Summary

- Never Assume Anything is secured
- Use Coding Standard
- Keep it simple
- Identify and evaluate the security of software used

References

Kerestan, B. (2017). How to detect, prevent, and mitigate buffer overflow attacks.