CYB2200 - Secure Coding And Analysis
Week 1 Notes

Course Notes

August 25, 2025

Contents

1 Lecture 1: Jia Song

1.1 Instructor Information

- Jia Song has PhD in Computer Science
- Email: Jsong@uidaho.edu (Preferred)
- Office: JEB 340
- In email use "[CYB2200] YOUR SUBJECT HERE" for subject
- MWF 9:30 - 10:20, labs on Friday

1.2 Why?

- So we can write robust code
- This class is required
- Find vulnerabilities before criminals

1.3 What?

1.3.1 SPP: Secure Programming Practices

- Principles of Secure Programming
- Robust Programming
- Defensive Programming
- Programming Flaws
- Static Analysis
- Different Programming Languages

1.3.2 SSA: Software Security Analysis

- Testing Methodologies
- Static and Dynamic Analysis Techniques
- Sandboxing
- Common Analysis Tools and Methods

1.3.3 QAT: QA/Functional Testing

- Testing Methodologies
- Test Coverage Analysis
- Automatic and Manual Generation of Test Inputs
- Test Execution

1.3.4 Bonus: STRIDE

S: Spoofing identity

T: Tampering with data

R: Repudiation

I: Information disclosure

D: Denial of service

E: Elevation of privilege

1.4 Deliverables

- Produce Software Security Analysis tools and techniques
- Apply knowledge to perform Software Security Analysis using common tools
- etc.

2 Lecture 2: Basic Security Concepts

2.1 Computer System, Computer Security, CIA Triad

2.1.1 Computer Security

- The protection of the items you value

2.1.2 Computer Systems

- Hardware
- Software
- Data

2.1.3 CIA Triad

Confidentiality: Ability of a system to ensure that an asset is viewed only by authorized parties

Integrity: Ability of a system to ensure that an asset can only be modified by authorized parties

Availability: Ability of a system to ensure that an asset can be used by any authorized parties

Computer security seeks to prevent unauthorized viewing (**confidentiality**) or modification (**integrity**) of data while preserving access (**availability**).

2.2 Vulnerability, Threat, Risk, Attack, Countermeasure

Vulnerability: A weakness in an information system, security system, procedures, internal controls, or implementation that could be exploited or triggered by a threat source [NIST]

Threat: A computing system is a set of circumstances that has the potential to cause loss or harm

Risk/Harm:
- Harm: negative consequence of an actualized threat
- Risk: Possibility of harm

Attack: A human that exploits a vulnerability is an attack on a system

Exploit: Is a piece of software, data, or sequence of commands that takes advantage of a vulnerability to cause unintended/unanticipated behavior to occur

Countermeasure: Things that prevent threats

2.2.1 Attack

An Attack is an attempt to gain unauthorized access to anything.

Passive attack: Attempt to collect, learn or use the info in the system, does not affect the system

Active attack: Attempt to alter system resources or change their operations

Insider attack: Attacks initiated by an insider who is authorized to access system resources

Outside attack: Attacks initiated by an outsider, usually an unauthorized user of the target systems

2.2.2 Controls/Countermeasures

Prevent threats from exercising vulnerabilities (before or after the fact):

- **Prevent** it, by blocking the attack or closing the vulnerability
- **Deter** it, by making the attack harder but not impossible
- **Deflect** it, by making another target more attractive
- **Mitigate** it, by making its impact less severe
- **Detect** it, by either as it happens or some time after the fact
- **Recover** from an attack's effects

2.3 Identification and Authentication and Access Control

Someone is authorized to take some action on something.

2.3.1 Someone

Who is the person?

Identification: The act of asserting who a person is

Identity: The set of physical and behavioral characteristics by which an individual is uniquely recognizable [NIST]

These can be public or well known or predictable:

- Email
- Student ID
- Employee ID

Authentication: The act of proving that asserted identity. Auth should be private and well protected.

2.3.2 Authentication Mechanisms

- Something the user **has**: A physical object (id card) in the person's possession
- Something the user **knows**: Passwords, SSN, PIN
- Something the user **is**: Fingerprints, retina, face, voice, etc.
- Something that uses more than one of these is a **MFA** or multi-factor authentication

2.3.3 Access Control

A subject is permitted to access an object in a particular mode, and only such authorized accesses are allowed.

Subject: Human users

Object: Things on which an action can be performed

Access Mode: Any controllable actions

Policies

- **Access control policies** indicate what types of access are permitted, under what circumstances, and by whom
- **Authorization** is the process of determining whether a user is permitted to perform a specific operation

2.4 Software Development Cycle

Software development life cycle describes phases of the software development cycle and the order in which those phases are executed. Each phase produces deliverables required for the next phase.

Security should be considered as early as possible:

1. **Planning**
2. **Analysis**

3. **Design**
4. **Implementation**
5. **Testing and Integration**
6. **Maintenance**

2.4.1 Terminology

Software Bugs: Errors, flaws, mistakes, or oversights in programs

Software Vulnerabilities: Specific flaws that allow attack vectors

Malware: Software that has mal-intent

3 Lecture 2: Continuing Basic Concepts

3.1 Robust Programming

A style of programming that prevents abnormal termination or unexpected actions.

- Handles bad inputs gracefully
- Detects internal errors and handles them gracefully

3.2 The Philosophy of Secure Programming

1. Remember what you have learned in the programming classes:
    - Check user input
    - Check your bounds
    - Assume an error will occur and handle it properly
        - What could someone deliberately do to compromise your program? (Adversary thinking)
        - What could someone unintentionally do to compromise your program? (People make mistakes)
2. Defensive Programming:
    - Input validation, type checking
    - Cover all cases - use defaults to handle cases not explicitly covered
    - Catch and handle exceptions at the lowest level possible
3. Understand the environment in which your program will be used:
    - Programs interact with people and with the system
4. Understand the procedures under which people will use your program:
    - The best program if installed incorrectly can compromise the system
    - The best program if configured incorrectly can also cause problems

3.3 How Do We Manage Software Vulnerabilities?

- Design and implement systems to avoid them
- Analyze and test systems to find them
- Add mitigation techniques to address them

3.4 Summary

- Computer security is the protection of the items you value, called the assets of an information system
- Confidentiality, integrity, and availability (CIA triad) are the three basic security objectives
- Computer security seeks to prevent unauthorized viewing (confidentiality) or modification (integrity) of data while preserving access (availability)
- Definitions: vulnerability, threat, control/countermeasure, harm, risk, attack
- Identification and authentication
- Authorization, Access control
- Software bugs/vulnerabilities, SDLC
- Robust programming, Defensive programming

4 Lecture 3: C-style Strings

4.1 Warm-up

- In C++:
  - Do you use arrays? — yes
  - Do you use strings? — yes
- In C:
  - The only way is char array

4.2 Why C?

- Developed in 1970 when security was not a concern
- Many common vulnerabilities
  - Some of the weak points do not exist in other languages, so we use C to get exposed to these
- Many legacy code running C
- Many existing software/systems were written in C/C++, which is still widely used

4.3 C-strings

Strings are not a built-in data type in C. In C, they are char arrays terminated by a NUL char (0x00).

```
int main()
{
    int scores[10];
    char name[100];
    int number_of_score = 0;
    double average = 0;
    int sum = 0;
    char grade = 'X';
    char comments[5] = "NONE";
}
```

Listing 1: Example C String Usage

4.4 Examples

```
char name[100] = "username";
cout << "size of name is: " << sizeof(name) << endl;
// output = 100
```

Listing 2: Example 1

```
char str[] = "hello";
cout << "size of str is: " << sizeof(str) << endl;
// output = 6
```

Listing 3: Example 2

```
char str2[5] = "hello";
cout << "size of str is: " << sizeof(str2) << endl;
// compiler warning
```

Listing 4: Example 3

When a char array is created, the null terminator is automatically added.

4.5 Two Major Problems with C Strings

1. **The length of the string and the size of the array**
   - If the string is bigger than the array we have a buffer overflow
2. **The NUL terminator**
   - NUL char is marking the end of a string
   - If it is missing, functions can continue reading chars

4.6 NULL != NUL

NUL: Null char, null terminator

- It is a char
- Indicates the end of a string char array

NULL: A macro

- Indicates a pointer doesn't have address

4.7 C Handling Vulnerabilities

Unsafe use of handful of functions:

Unbounded string Functions: (example: copy function)

- The destination buffer's size isn't taken into account at all
- Buffer overflow (source data's length exceeds the destination's buffer size)

Bounded String functions:
- Safer option to unbounded
- They handle lengths

4.8 Printf() in C

- `%d` = decimal
- `%x` = address

```c
printf("hello world!\n");
printf("a has value: %d\n", a);
printf("a has value: %d\n, b has value: %d\n", a, b);
printf("a has value: %d\n, b has value: %d\n, c is at address: %x\n", a, b, &c)
    ;
```

Listing 5: Printf Examples

`int printf(const char *format, ...)`
The `...` indicates that zero or more optional args can be provided.

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

Table 1: Printf Format Specifiers

4.9 Unbounded String Functions

4.9.1 scanf() - Reading Input

Function: `int scanf(const char *format, ...)`

Purpose: The `scanf()` function parses input according to the format argument.

```c
#include <stdio.h>

int main() {
    char name[20];
    scanf("%s", name);
    printf("Your name is: %s", name);
    return 0;
}
// This program demonstrates the use of scanf() to read a string input.
// It is important to ensure that the input does not exceed the buffer size.
// scanf() can be used with a maximum field width to prevent buffer overflow.
```

Listing 6: scanf Example

4.9.2 sprintf() - Reading Input

Function: `int sprintf(char *str, const char *format, ...)`

Purpose: The `sprintf()` functions print a formatted string into a destination buffer.

```c
#include <stdio.h>

int main() {
    char buffer[20];
    int a = 5, b = 3, k;

    k = sprintf(buffer, "%d plus %d is %d", a, b, a+b);
    printf("[%s] is a string, its length is %d. \n", buffer, k);

    return 0;
}
// "5 plus 3 is 8", is a string, its length is 13
```

Listing 7: sprintf Example

5 Metadata/Metacharacters

5.1 Metadata

Metadata accompanies the main data and provides info about it.

5.1.1 Problems

- **Embedded delimiters**: Delimiters are used to denote the termination of a field
- **Truncation**

5.2 Filtering

Three options:

1. Detect erroneous input and reject what appears to be an attack
2. Detect and strip dangerous characters
3. Detect and encode dangerous character with a metacharacter escape sequence

5.3 Eliminating Metacharacters

Reject illegal requests:
- Any request containing illegal metacharacters is simply discarded; processing terminates
- Fewer things can go wrong in the handling
- The application may be unfriendly

Strip dangerous characters:
- Filters modify the input to get rid of any violations
- Filters need to be implemented carefully
  - Blacklist (like a ban)
  - Whitelist (only some allowed)

Encoding metacharacters (escaping):

6 Types

6.1 Typed vs Untyped Languages

- C/C++: Strictly typed
- JavaScript/Python: Untyped

6.2 Data Types

6.2.1 Primary

- Integer
- Char
- Bool
- Floating Point
- Double floating point
- void
- wide char

6.2.2 Derived

- Function
- Array
- Pointer
- Reference

6.2.3 User Defined

- class
- struct
- union
- enum
- typedef

6.3 Why Do We Need Types?

- In our eyes, it's easy to remember what stuff is:
  - `int i = 90`
  - `double k = 4.90`
  - `char c = 'c'`
- But for the compiler, types are used to determine memory space
- It also tells the compiler how to process the data

6.4 Data Storage

C data types:

char, signed/unsigned char: 1 byte of storage

Int:
- short int — 2 bytes
- int — 4 bytes
- long int — 4/8 bytes
- long long int — 8 bytes

Floats:
- float — 4 bytes
- double — 8 bytes
- long double — 12 bytes

6.4.1 Signed Ints vs Unsigned Ints

- 1 byte = 8 bits
- 8 bits can hold 0–255 unsigned
- 8 bits can hold −128 to 127 signed

6.5 Int Representation

6.5.1 Signed and Magnitude

- The sign of the number is stored in the sign bit
- 0 → positive
- 1 → negative
- Problem: 00000000 = positive 0, 10000000 = negative 0 (invalid)

6.5.2 Two's Complement

- The sign bit is 1 if the number is negative and 0 if the number is positive
- Positive values can be read directly from the value bits
- Negative values can't be read directly; the whole number must be negated first
- In ones complement, a number is negated by inverting all its bits

7 Integer Overflow/Underflow

[Note: This section was indicated in the original notes but not detailed]

8 Problems with Type Conversions

8.1 Type Conversion

We want to convert one data type to another:

Explicit type conversions: The program knows

Implicit type conversions: The compiler does it behind the scenes

- Conversions can lead to lost or misinterpreted data
- Conversion rules — the general rules C uses when converting between types:
  - Simple conversions
  - Integer promotions
  - Arithmetic conversions

8.2 Conversion Rules

8.2.1 Simple Conversions

1. Casts

```c
int age; // is a cast

```

2. Assignment statements

```c
short int fred;
int bob = -10;
fred = bob;

```

3. Type conversion function arguments

```c
int dostuff(int num, unsigned int length);

void func(void) {
    char a = 42;
    unsigned short b = 43;
    long long int c;

    c = dostuff(a,b); // vars are converted to the needed type
}

```

4. Type conversions function returns

```c
char func(void)
{
    int a = 42;
    return a; // need char, a = int, so type converts
}

```

8.2.2 Simple Conversion Rules

Value-preserving conversion: If the new data type can match all possible values of the old type, the conversion is said to be value preserving

Value changing conversion: The old type can contain values that can't be represented with the new

8.3 Int Types

8.3.1 Widening

Converting from a narrow type to a wider type.

The machine typically copies the bit pattern from the old var to the new one, then fills the rest of the value bits with 1s or 0s depending on the type:

Zero extension: If the source is unsigned, propagates the value 0 to all high bits

Sign extension: If the source is signed, propagates the sign bit to all high bits

8.3.2 Narrowing

When converting from a wider data type to a narrower type, the machine uses only one mechanism: truncation.

8.4 Int Conversion Errors

Example:

- `juice = apple + orange` (valid)
- `if (apple > orange)` (??? what do we compare?)

Hence: hidden type conversions → transform both operands into a comparable data type.

8.4.1 When Does This Happen?

- Arithmetic: `+, -, *, /, %, ...`
- Relational/equality: `<, >, <=, >=, ==, !=`
- Bitwise: `&`, etc.

8.5 Integer Promotions

- Int types smaller than int are promoted when an operation is performed on them
- Promoted types (if result is int):
  - unsigned/signed char
  - unsigned/signed short

8.6 Conversion Rules

8.6.1 Rule 1: Floating Points Take Precedence

- If one arg is a floating point, the other arg is converted
- If one floating point is less precise than the other, it is converted to be more precise