COMS W4156 Advanced Software Engineering (ASE)

October 25, 2022

Agenda

1. (In)Secure Coding

Static Analysis vs. Testing for Detecting Security Vulnerabilities

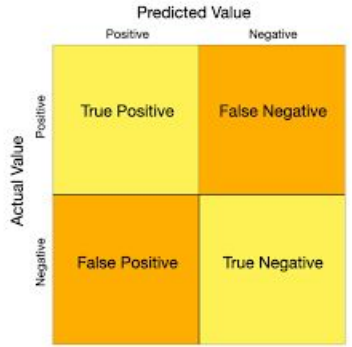
Testing (and any dynamic analyses that monitor code execution) finds bugs that *did* happen with a specific input, but it's infeasible to consider all possible inputs so misses many bugs - false negatives

Static analysis finds code patterns that *might* lead to bugs and bugs that *might* happen with some inputs - false positives

Is this a false positive or a real bug?

```
function foo (int x) {
   if (x<0) { do something buggy }
   else { do something not buggy }
```

```
function bar (int y) {
   if (y<0) return;
   foo(y);
}
```



Writing Code the Right Way: Secure Coding

Static analysis tools look for “patterns” in the code (or in an internal representation of the code, not necessarily the source code text)

They flag code that does not conform to required patterns (*always do*) or matches anti-patterns to be avoided (*never do*)

Some patterns and anti-patterns are specifically concerned with security, e.g., checking for anti-patterns corresponding to common weaknesses and vulnerabilities

What is a (Software) Security Vulnerability?

A bug that someone has figured out how to exploit

A bug that someone has not yet figured out how to exploit

"*Software flaws are security flaws*" (Rebecca Wright)

Check for all errors and exceptions

Hackers can exploit a software vulnerability by crafting invalid input that is not caught by input validity checks, and then that invalid input is processed as if it were valid. So it's crucial for API documentation to clearly define all status codes and exceptions and for calling code to handle every status code and exception that could possibly be returned/raised by an API call

https://cwe.mitre.org/ -> "handling status codes"

Long list of Linux error codes: errno - number of last error

Example

```
status = read(input);

if (status = tooLong OR status = tooShort)

    reject input;

else

    process input;
```

possible status codes returned from read include:

input too long

input too short

input is evil 

input just right

Errors vs Exceptions

- A user entering the wrong data is not exceptional and does not need to be handled with an exception. Simple checks at both front-end and back-end can address user errors and show meaningful error messages. But never rely on the client to protect the server - the hacker controls the client!

- A file won't open (and the server code throws FileLoadException, FileNotFoundException, etc.) is an exceptional situation that should be handled by catching the exception - and show meaningful error messages to the user

➢ If the "user" is - or could be - client code, rather than a human, then the service must return meaningful status codes according to an agreed-upon protocol and the user/caller code should branch on status codes

Input Validity Checks

The third-party code that your code calls might validate all data from untrusted sources and return a corresponding status code …
or it might not

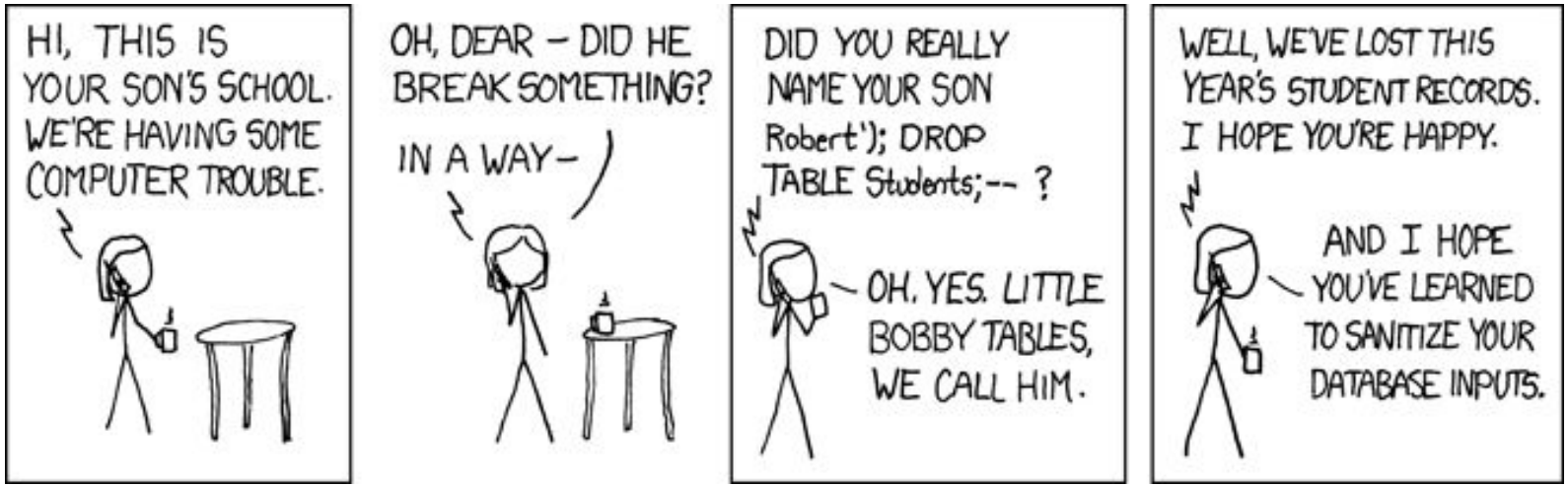
What is wrong with this code snippet? What should be checked but isn't?

https://cwe.mitre.org/ -> "improper validation"

```
public static final double price = 20.00;

int quantity =
currentUser.getAttribute("quantity");

double total = price * quantity;

chargeUser(total);
```

Unsanitized User Inputs

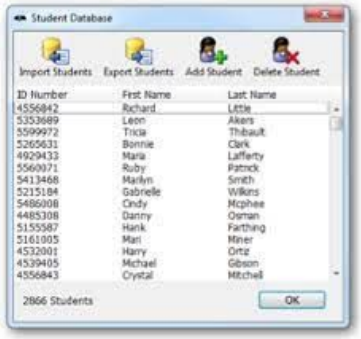
When user inputs are passed to database, API, logger, external system, etc.



Explanation

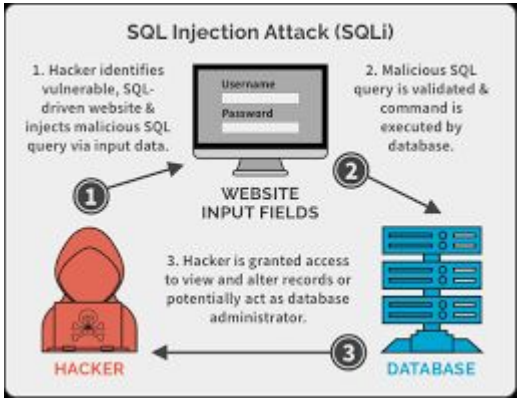
The school apparently stores student names in a database table called Students. When a new student enrolls, the school inserts his/her name into this table. The code doing the insertion might look like:

```
$sql = "INSERT INTO Students (Name)
        VALUES ('" . $studentName . "');";
execute_sql($sql);
```



This code first creates a string containing an SQL INSERT statement. The content of the $studentName variable is glued into the SQL statement. Then the code sends the resulting SQL statement to the database. Untrusted user input, i.e., the content of $studentName, becomes part of the SQL statement.

Explanation continued



Say the user input is “Sarah”, then the SQL is:

INSERT INTO Students (Name) VALUES ('Sarah')’

This inserts Sarah into the Students table.

Now say the user input is “Robert'); DROP TABLE Students;--” then the SQL statement becomes

INSERT INTO Students (Name) VALUES ('Robert');

DROP TABLE Students;--');

This first inserts Robert into the Students table. But since the INSERT statement is followed by a DROP TABLE statement , next the entire Students table is deleted.

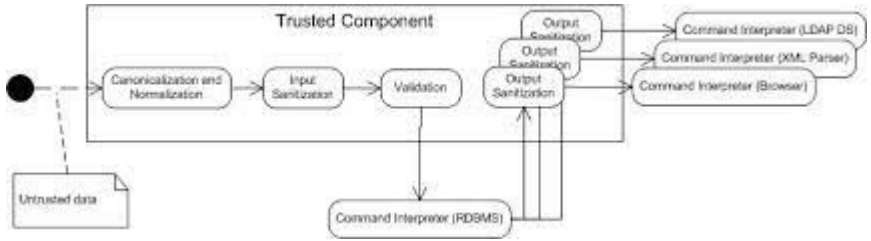
Sanitizing User Inputs

However, what it means to “sanitize” user inputs is not well-defined

The only guaranteed way to avoid an “SQL injection” attack is to use “prepared statements” or “parameterized statements”

Every programming language commonly used with databases has some facility for passing user inputs as parameters to prepared SQL statements

Example



Unsanitized User Input without SQL

```
if (loginSuccessful) {
  logger.severe("User login succeeded for: " + username);
} else {
  logger.severe("User login failed for: " + username);
}
```

Say user has entered username “guest”, the log gets

```
May 15, 2020 2:19:10 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: guest
```

How To Sanitize?

Say the user entered username
“guest
May 15, 2020 2:25:52 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login succeeded for: administrator“

The log gets

```
May 15, 2020 2:19:10 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: guest
May 15, 2020 2:25:52 PM java.util.logging.LogManager log
SEVERE: User login succeeded for: administrator
```

[A1:2017-Injection]

Vulnerability Lifecycle

https://cwe.mitre.org/ common weakness enumeration
https://cve.mitre.org/ cybersecurity vulnerability enumeration

What are the Ultimate Goals of Secure Coding?

Confidentiality: Data not leaked to unauthorized users, entities or processes

Integrity: Data not lost, changed in transit, or modified by unauthorized users

Availability: Data and system functionality is consistently accessible when needed by authorized users

(“CIA triad”)

Authenticity: Data origin cannot be spoofed

What is Insecure Coding?

Code likely to lead to CIA or Authentication breaches

Simplest example: secret keys, tokens or passwords embedded in code

String constants containing domain-specific patterns - in this case access key patterns known to appear in AWS configurations

[CWE-798: Use of Hard-coded Credentials]

Tree: 6d33b... / [redacted] / aws / claudia / **server.js**

[redacted]aws_exercises

1 contributor

21 lines (15 sloc) | 361 Bytes

```
1  var ApiBuilder = require('claudia-api-builder'),
2      api = new ApiBuilder();
3
4  module.exports = api;
5
6  AWS.config.update({
7    "accessKeyId": "AKIA[redacted]",
8    "secretAccessKey": "[redacted]",
9  }
10 );
```

What is Wrong?

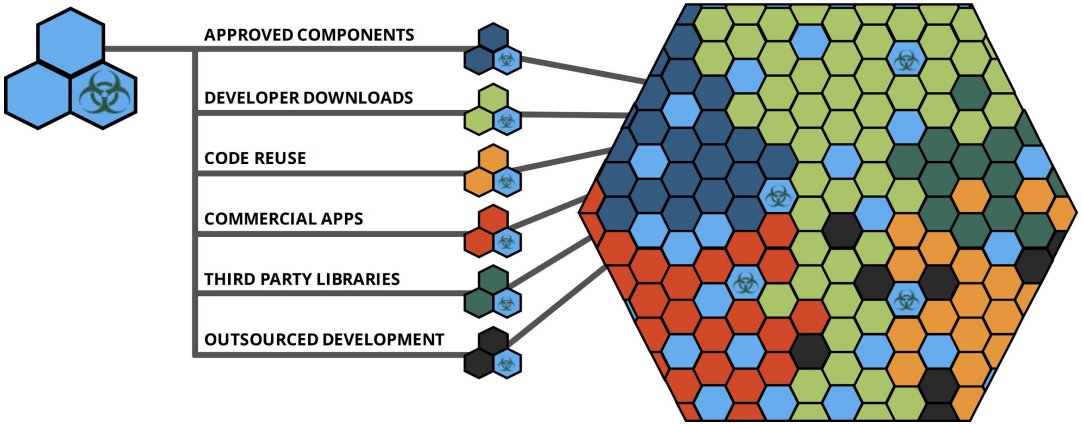
Found in a tutorial on how to add password protection to a page using Javascript

CWE-798: Use of Hard-coded Credentials

Using Third-Party Libraries with Known Vulnerabilities

Detected by comparing code resources to databases of known security vulnerabilities

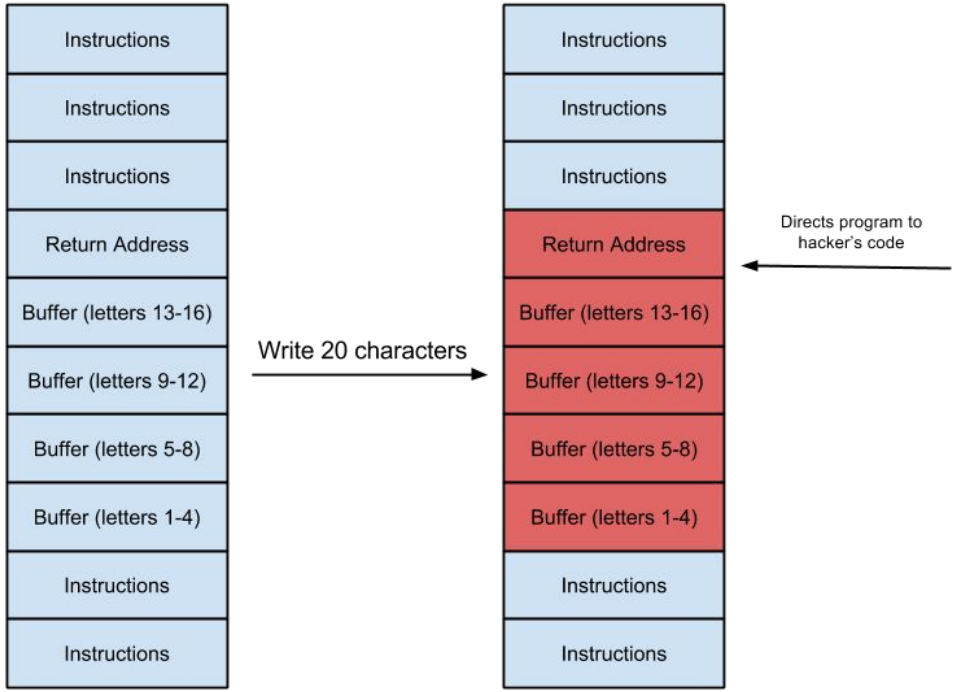
[A9:2017-Using Components with Known Vulnerabilities]



Everyone’s Favorite Vulnerability: Buffer Overflows

A bug finder for unmanaged languages (notably C/C++) should look for code patterns that check array bounds, to prevent buffer overflows

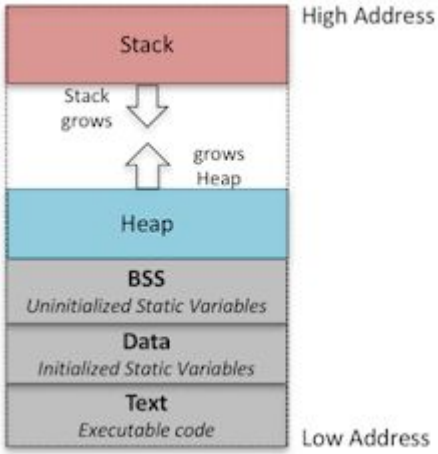
https://cwe.mitre.org/ -> “buffer overflow”



Memory Allocation

Bug finders for unmanaged languages should also check that every memory allocation is followed (in every execution path) by a corresponding memory deallocation, and there are no further uses of the memory after it has been deallocated

https://cwe.mitre.org/ -> “malloc free”



Dereferencing a Null Pointer

```
int a, b, c; // some integers
int *pi;     // a pointer to an integer

a = 5;
pi = &a; // pi points to a
b = *pi; // b is now 5
pi = NULL;
c = *pi; // this is a NULL pointer dereference
```

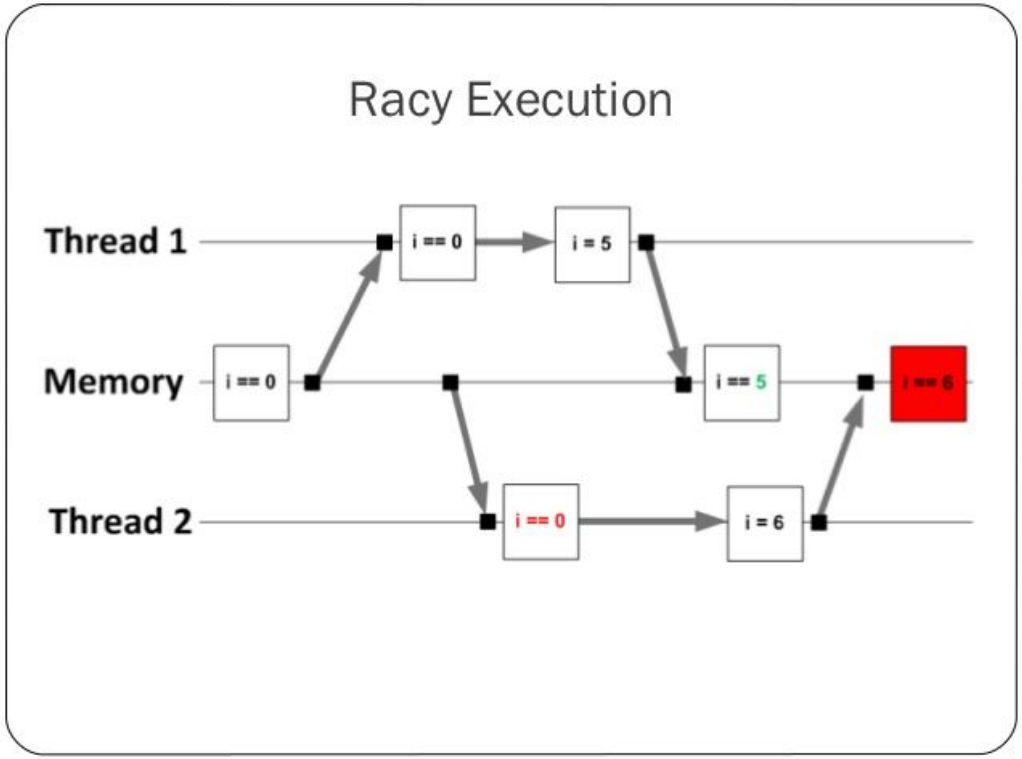
```
int a, b, c; // some integers
int *pi;     // a pointer to an integer

a = 5;
pi = &a; // pi points to a
b = *pi; // b is now 5
pi = abracadabra(a, b);
c = *pi; // could pi be a NULL pointer?
```

[You're dereferencing a null pointer!]

Concurrency Bugs

Some static analysis bug finders check multi-threaded code, e.g., matching locks with unlocks along every execution path and detecting possible data races

https://cwe.mitre.org/ -> “lock unlock”, “race condition”



Suppressing Warnings

SonarQube is probably the best multi-language bug finder, but you need to set up filters or drown in warnings

Static analysis bug finders typically produce many *false positives*, e.g., reporting a missing free or unlock on an execution path that is actually infeasible at runtime. So it is necessary to be able to *suppress* certain error/warning messages. For example, once a developer has marked a particular warning as a **false positive**, the bug finder should never give that same warning for the same code or "similar" code

Static bug checkers cannot avoid *false negatives*, they cannot find "all" bugs and cannot "prove" no bug exists. Static analysis does not replace dynamic analysis, particularly testing - static analysis finds bugs that *might* happen with some input. Testing finds bugs that *did* happen with a specific input, but it's infeasible to consider all possible inputs for non-trivial programs

Secure Coding Practices

Bad Coding Practices

API / Function Errors

Secure coding guidelines: secure coding in Java , secure coding in C++

What is the Easiest Way to Secure a Computer

Disconnect it from the network

Disable all access to external media (e.g., no flash drives, no USB)

Turn it off and disconnect all power sources

Wipe its disks and non-volatile memory

Upcoming Assignments

First Iteration Demo due October 31

➔ You can keep coding and testing between first iteration submission and first iteration demo, but tag both separately! Also see github releases

First Individual Assessment
available 12:01am November 1, due 11:59pm November 4

Next Time

Bug Finder demo

Design Principles

Ask Me Anything