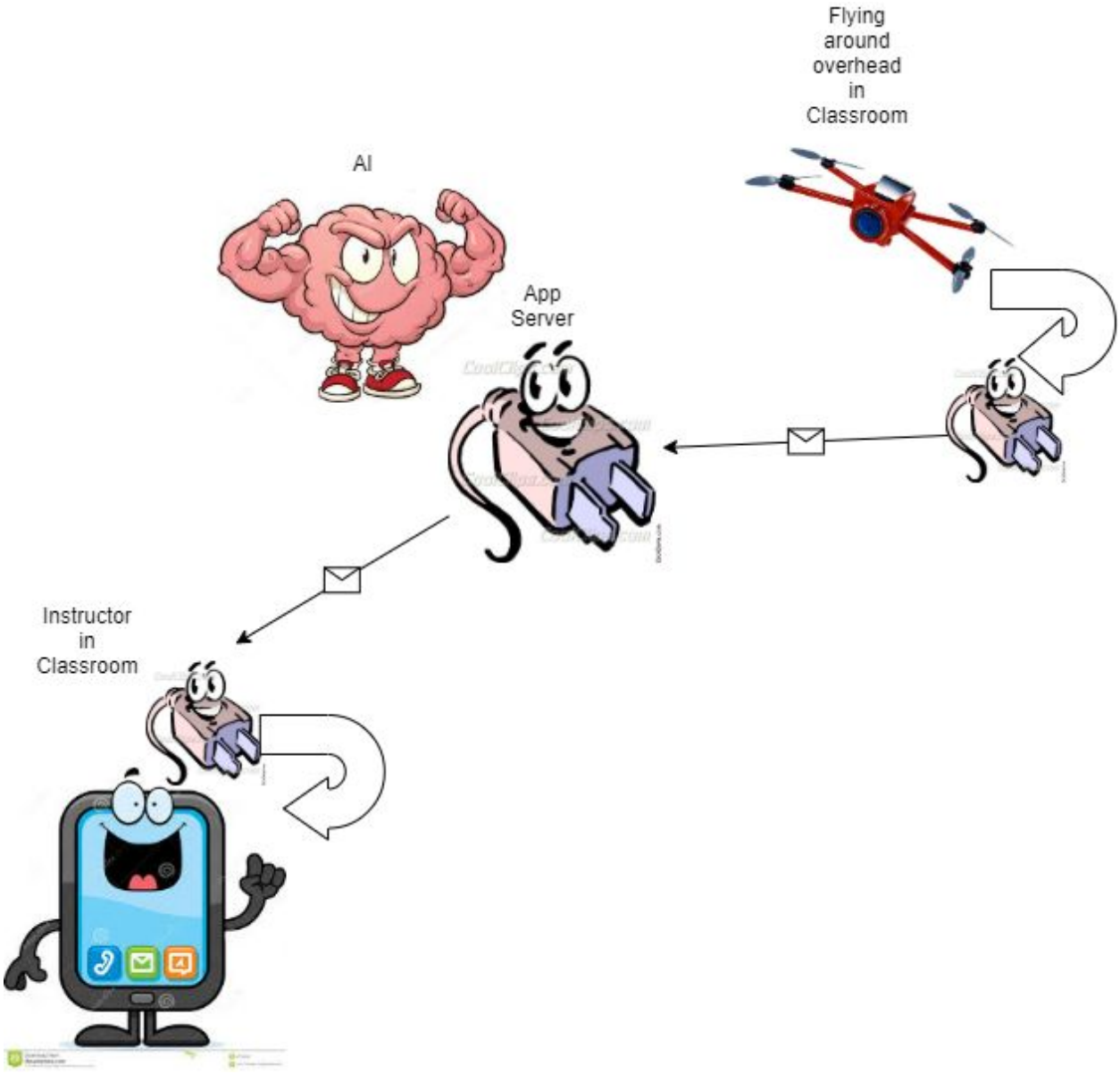
Lecture Notes
October 22, 2020

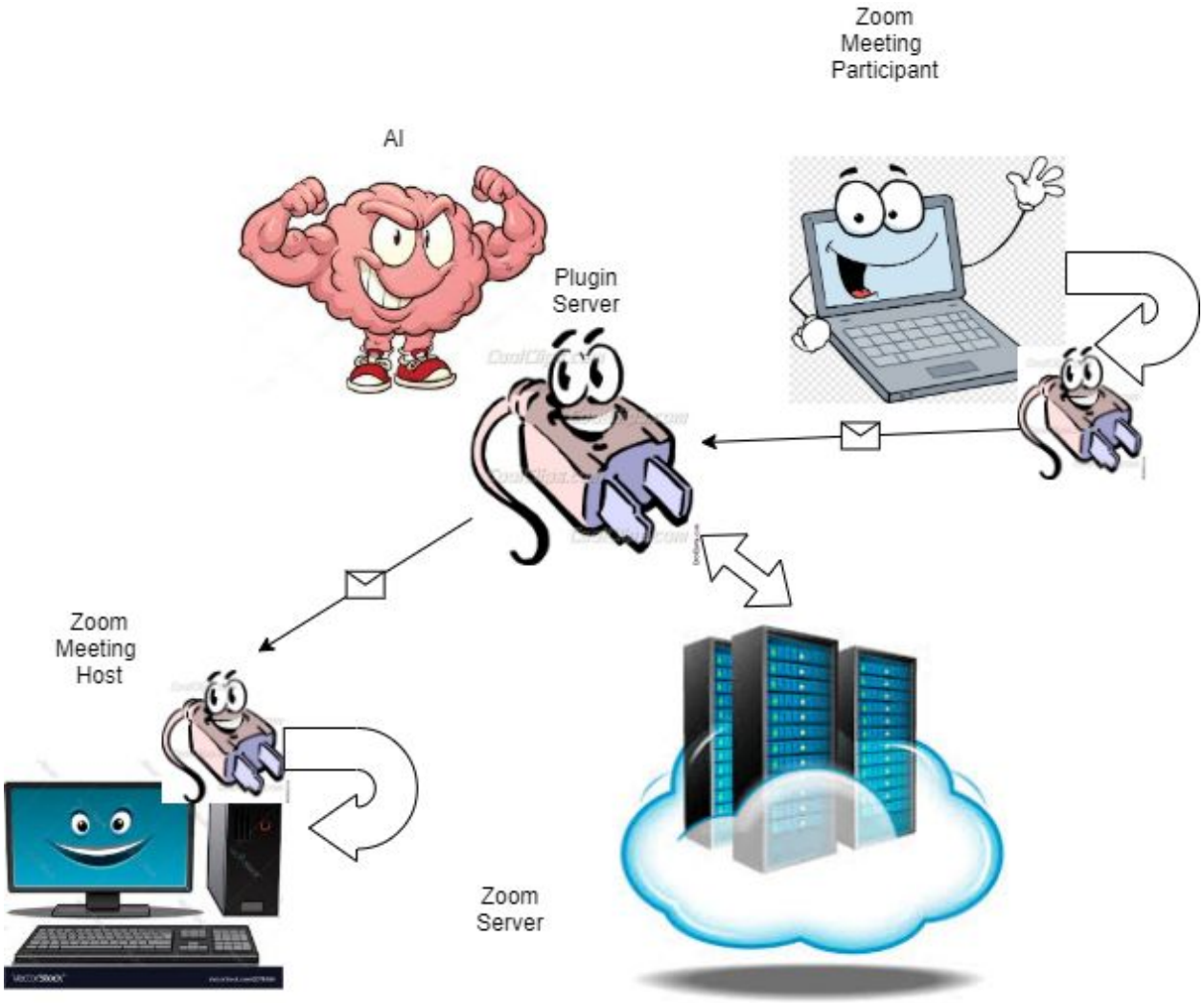
Discuss first assessment
https://courseworks2.columbia.edu/courses/104335/assignments/496168

First question: Write use cases corresponding to the code you just worked on for about a month

Second question: Sketch an API and user stories for functionality similar to the (x3) breakout room exercise

Use cases and user stories presented in lectures, APIs presented in lectures and first individual assignment


Flying
around
overhead
in
Classroom
AI
App
Server
Instructor
in
Classroom


Zoom
Meeting
Participant
AI
Plugin
Server
Zoom
Meeting
Host
Zoom
Server

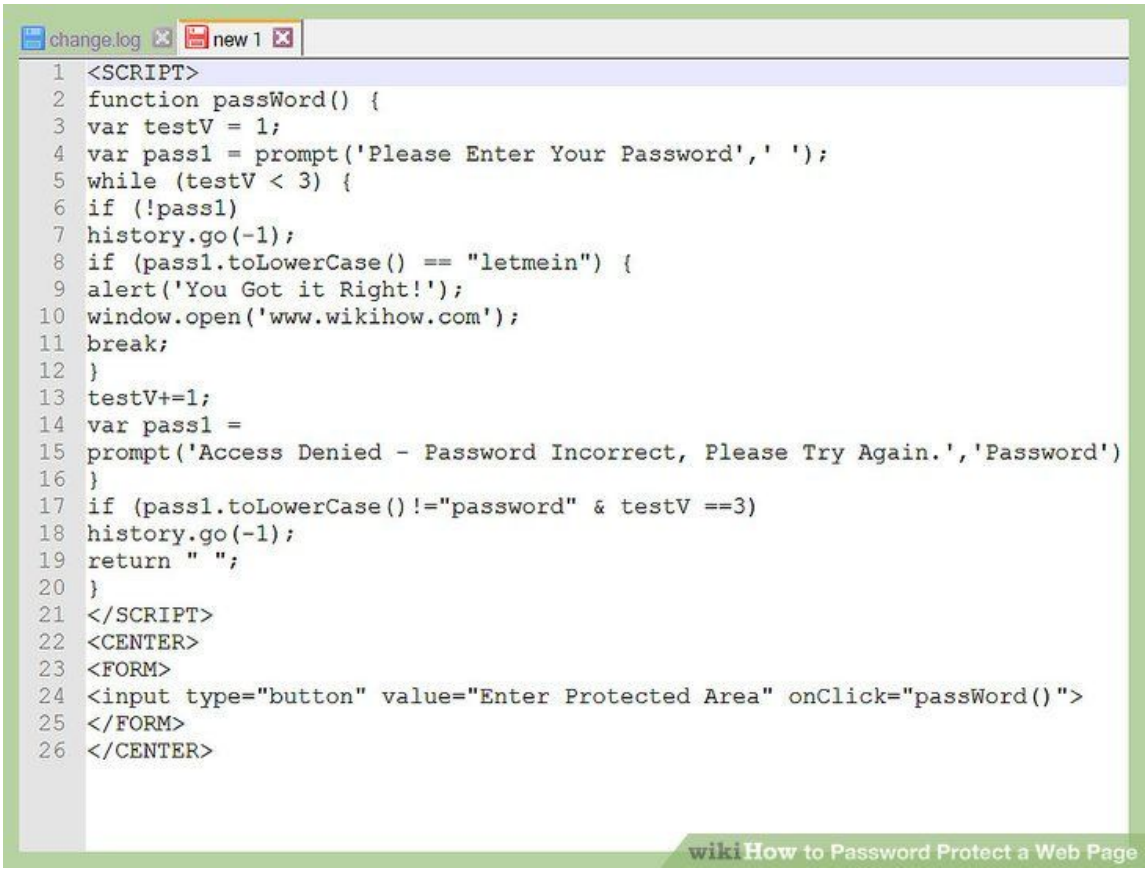
Let’s discuss some more examples of problems, particularly security *anti-patterns*, that can be found by static analysis bug finders:

Secret keys, tokens or passwords embedded in code

The bug finder looks for 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]

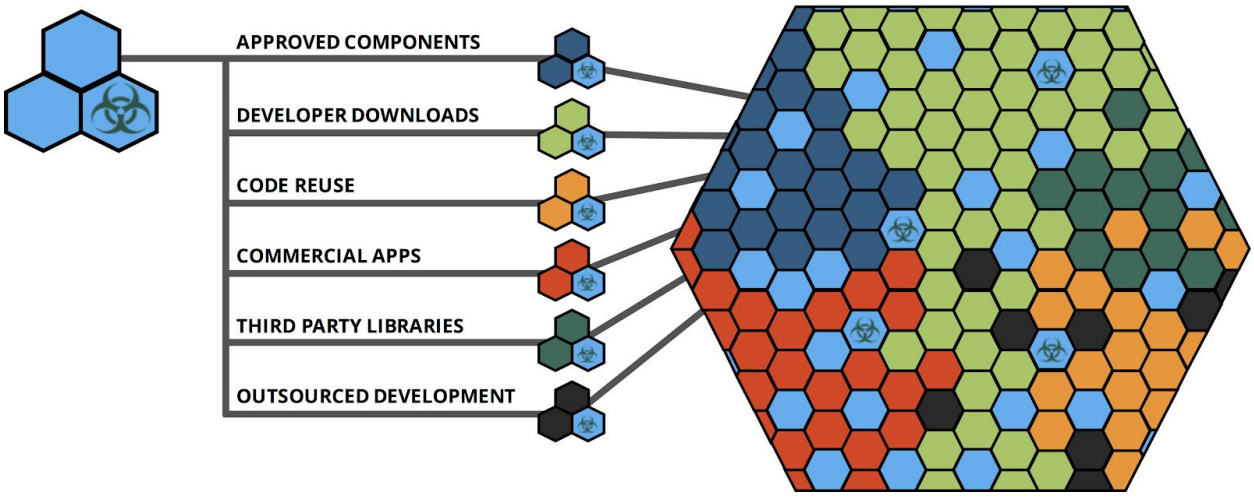
The following was found in a tutorial on how to add password protection to a page using Javascript. What is wrong here?



Hint: right click page -> view source
for runtime details, ctrl-shift-i (Win) or cmd-opt-i (Mac)

CWE-798: Use of Hard-coded Credentials

Some bug finders can spot when applications use third-party libraries with known security vulnerabilities



The bug finder tool compares code resources to databases of known security vulnerabilities

[A9:2017-Using Components with Known Vulnerabilities]

Bug finders can check whether client code handles every error code or exception from each library or system call

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

Errors vs. exceptions:

- A user entering the wrong data is not exceptional and does not need to be handled with an exception. Simple checks, on both front-end and back-end, can address user errors - and show meaningful error messages to the user

- A file won't open and is throwing FileLoadException or FileNotFoundException. This is an exceptional situation that should be handled by the application by catching the exception with appropriate processing code - and, again, show meaningful error messages to the user

➢If the "user" is other code, then send meaningful status codes according to an agreed-upon protocol, and the client code should branch on status codes

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

Some bug finders look for dereferences of a null pointer

```
int a, b, c; // some integers
int *pi;     // 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
```

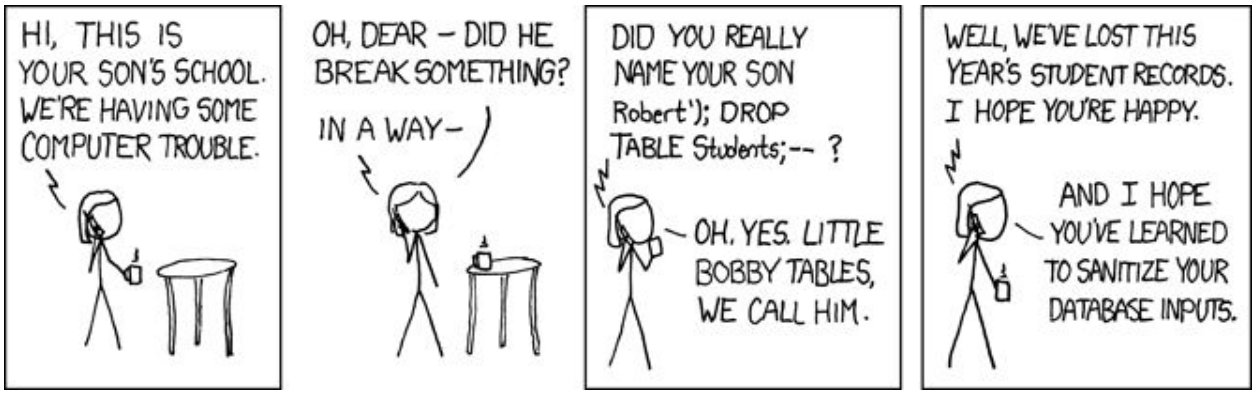
Static analysis can check whether it's possible for a pointer variable to be set to NULL on any path to a given statement that dereferences the pointer. In this case there's only one simple path.

What should the bug finder do in the following case?

```
int a, b, c; // some integers
int *pi;     // 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?
```

CWE-476: Null Pointer Dereference

Some bug finders can detect when user inputs are passed to database, API, logger, external system, etc.



The school apparently stores student names in a database table called Students. When a new student arrives, 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.

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 inserts Robert into the Students table. But the INSERT statement is followed by a DROP TABLE statement - which removes the entire Students table.

However, what it means to “sanitize” user inputs is not well-defined. The only way to avoid this kind of 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

Injection attacks are not restricted to SQL databases:

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

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

October 22, 2020 10:57:10 AM

java.util.logging.LogManager$RootLogger log

SEVERE: User login failed for: *guest*

Say the user entered username

“guest

April 1, 2021 11:59:59 PM

java.util.logging.LogManager$RootLogger log

SEVERE: User login succeeded for: administrator“

The log gets

October 22, 2020 10:39:10 AM

java.util.logging.LogManager$RootLogger log

SEVERE: *User login failed for: guest*

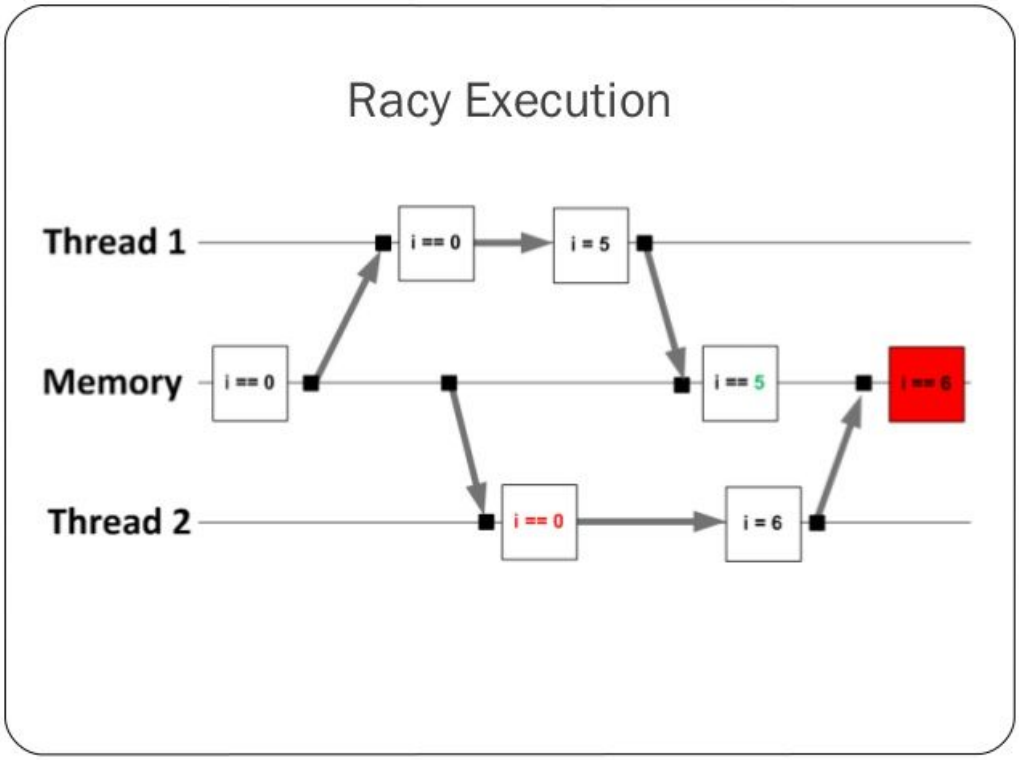
April 1, 2021 11:59:59 PM

java.util.logging.LogManager$RootLogger log

SEVERE: User login succeeded for: administrator

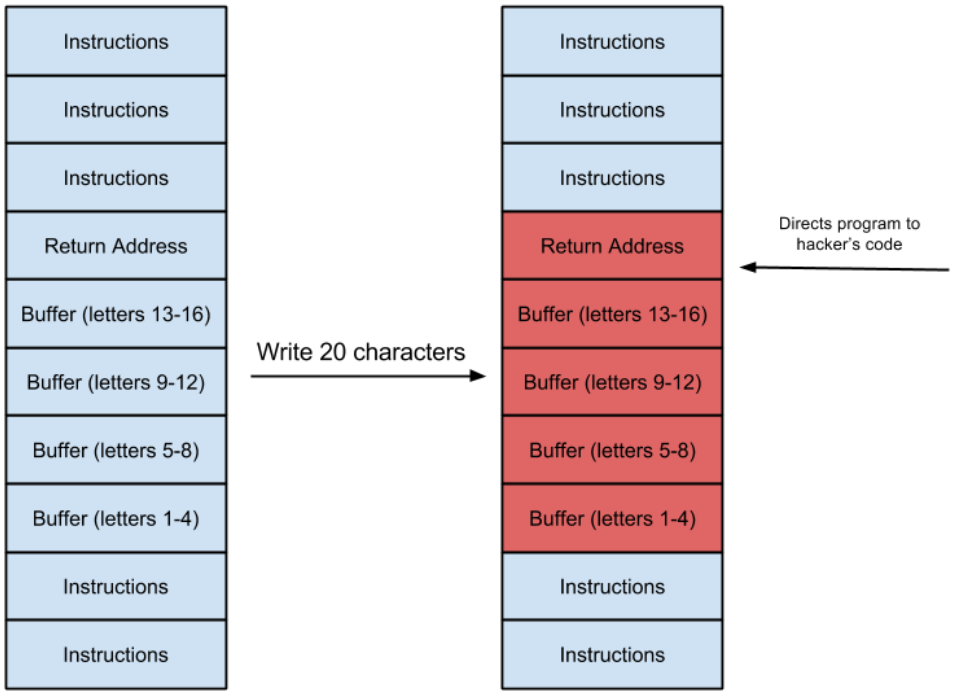
A1:2017-Injection

Some 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”

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



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

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”

Many other problems can be detected with static analysis

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

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

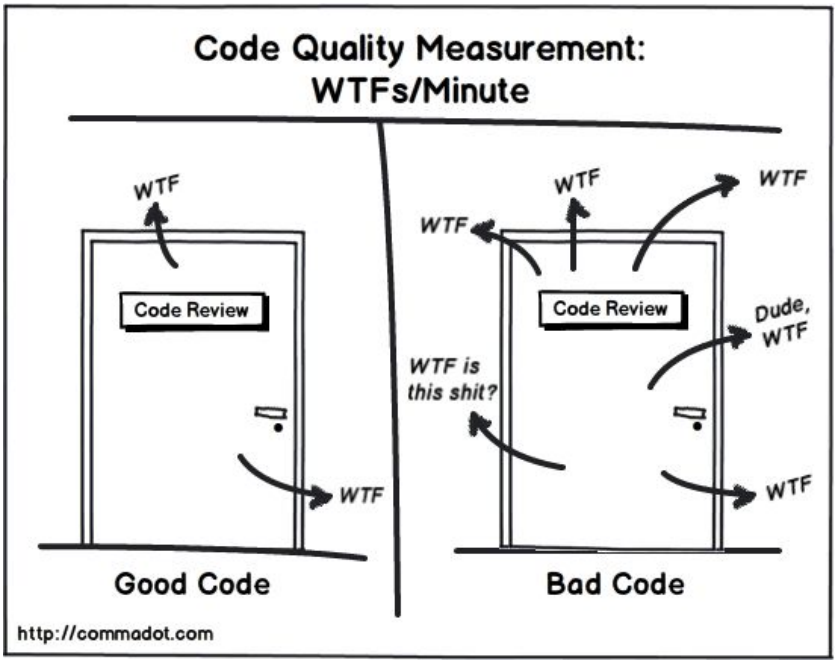
Static analysis does not replace dynamic analysis, particularly testing - static analysis finds bugs that *might* happen with some input. Testing finds bugs that *did* happen with a specific input, but it’s infeasible to consider all possible inputs for non-trivial programs

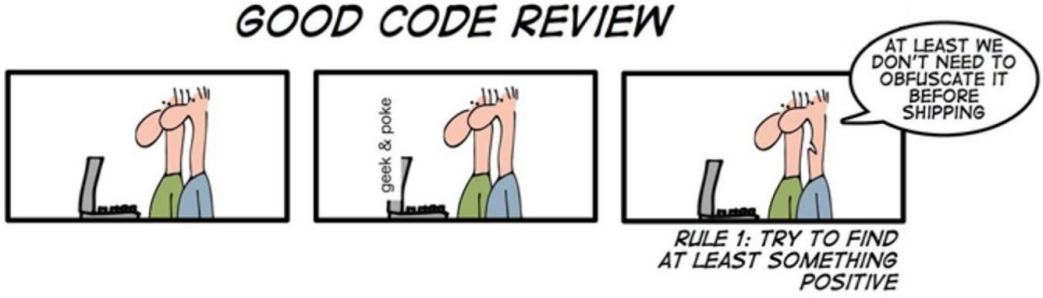
Code review = *human* style checker + bug finder

Static analyzers are tools that automatically detect style violations, code smells, security vulnerabilities, and generic bugs. In “code review”, one or more *humans* read and analyze the code

Like automated static analysis, code review can potentially consider *all* inputs whereas testing can only consider *some* inputs - probably a very small subset of all inputs.

Code review can find problems that dynamic testing and static analyzers cannot, such as code that passes style checking and other static analysis, and passes all test cases, but is still unreadable. Humans reviewers can also spot application-specific bugs as well as generic bugs that are beyond the state of the art in static analysis


Code Quality Measurement:
WTFs/Minute
WTF
Code Review
WTF
Good Code
WTF
WTF
WTF
WTF is this shit?
Code Review
Dude, WTF
WTF
Bad Code
http://commadot.com


HOW TO MAKE A GOOD CODE REVIEW
geek & poke
AT LEAST WE DON'T NEED TO OBFUSCATE IT BEFORE SHIPPING
RULE 1: TRY TO FIND AT LEAST SOMETHING POSITIVE

Instant code review happens on the fly during pair programming, where the navigator reviews while the driver writes/edits

Code review helps train junior developers - over-the-shoulder synchronous review by senior developer

Many organizations require code to be reviewed by a peer, a senior developer, or a designated reader prior to committing to the shared code repository

Lightweight forms of code review involve one or two readers besides the coder

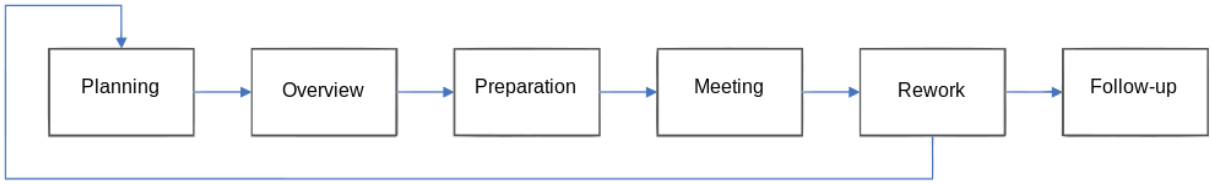
May be synchronous (reviewer sits with coder immediately after coder “done”) or asynchronous (separately). Aside from pair programming and junior developer training, there are major problems with synchronous reviews, so asynchronous generally considered better

Formal code review (sometimes called Fagan inspection) is heavyweight - involves a *team* review process

Predefined roles:

- Moderator - runs meeting, often from outside development team such as another team or a separate quality assurance group
- Recorder - takes notes
- Reader - sometimes author, sometimes intentionally not the author (who may not find code intent as obvious as the author). Reads the code aloud, walkthrough line by line, explain along the way
- Everyone else is called an Inspector - attendees might include customer representatives, end-users, testers, customer/technical support, etc.

Formal inspection proceeds in several steps. Check the output of each step and compare it to the desired outcome. Decide whether to move on to the next step or still have work to do in the current step. Meeting <= 2 hours.



Planning: Prepare materials - code should already compile, pass static analysis and unit tests (don't waste human time for work that can be automated), Arrange participants and meeting place

Overview: Group education of participants on the materials under review, Assignment of roles

Preparation: The participants review the item to be inspected and supporting material to prepare for the meeting noting any questions or possible defects

Inspection meeting: Walkthrough, Problems classified as Minor - developer is trusted to fix on own; Moderate - moderator checks; Severe - need another review meeting

Rework: Defects found during the inspection meeting are resolved by the developer.

Follow-up: For moderate or severe problems, moderator verifies or there's another review meeting, respectively

Mandated for safety-critical software, or any software that must work right the first time and every time, very rarely used for conventional business and consumer software other than for training purposes

“Assignment T1: Preliminary Project Proposal” due October 27
https://courseworks2.columbia.edu/courses/104335/assignments/486922

Each team proposes their own project, within constraints:

- Must impose authenticated login.
- Must be demoable online (e.g., zoom or discord).
- Must store some application data persistently (database or key-value store).
- Must use some publicly available API beyond those that “come with” the platform. It does not need to be a REST API.

Describe Minimal Viable Product (MVP) both in prose - a few paragraphs - and via 3-5 “user stories”.

< label >: As a < type of user >, I want < some goal > so that < some reason >.

My conditions of satisfaction are < list of common cases and special cases that must work >.

Describe acceptance testing plan that covers these cases.

List the tech you plan to use. If different members of the team plan to use different tools, please explain.

Two sample team projects from a previous offering of this course are linked below. All submitted assignments as well as the code are included in each repository. These projects had three iterations because the team project ran the entire duration of the course, there was no individual project.

Code Phoenix - https://github.com/s4chin/coms_4156

Space Panthers - https://github.com/wixyFun/openSpace