

# COMS W4156 Advanced Software Engineering (ASE)

December 2, 2021

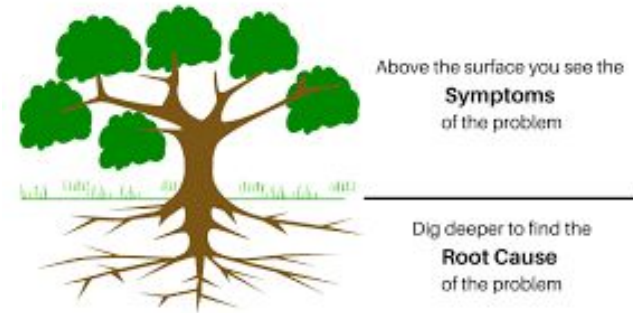
[shared google doc for discussion during class](#)

# Localizing Bugs

Testing *detects* bugs. Before you can fix a bug (by editing the code), you need to *localize* the bug (to know which lines to edit). You know it exists, but where is it? What is the “root cause”?

Do not confuse with application [“localization”](#), which concerns adaptation of UI and content to language, culture and locale (usually in an international context)

Localizing bug = find coding mistake  
in specific line(s) of code



# Where is it?



Unit testing makes it easier to localize bugs, since if a fault manifested during a unit test, then it's likely in that unit

Or supposed to be in that unit but omitted or unreachable - this is an advantage of [test-driven development](#) (TDD), where you write tests for what the code *is supposed to do* before writing the code rather than writing tests to check *what the code does* after writing the code, but localizing bugs usually refers to logical errors in reachable code

Coverage tools can often tell the developer exactly which statements and branches were exercised by the failing test(s)

Embedded assertions that check application state and logging statements that record application state also help pinpoint the flaw

# You Believe Your Code Is Correct

Localizing a bug is a process of checking beliefs about the program

- Developer might believe that a function was called with particular parameters
- Developer might believe that a certain variable has a certain value at a certain place in the code
- Developer might believe that a system or library call returned a specific value
- Developer might believe that a particular execution path is taken through conditionals

You know what your beliefs are while writing the function, so add these beliefs to each function while writing it using assertions and logging

Comments help but assertions/logs are better



# Code Beliefs

```
int foo (int a, int b) {  
    int c;  
    ...  
    code does something with a and b  
    ...  
    c = external_call(a,b);  
    ...  
    code does something with a, b and c  
    ...  
    return c;  
}
```

```
int foo (int a, int b) {  
    int c;  
    [ believe a>0 and b>0 ]  
    ...  
    [ believe a>0 and b>0 ]  
    c = external_call(a,b);  
    [ believe a>0 and b>0, believe c != 0 ]  
    ...  
    [ believe a>0 and b>0, believe c != 0 ]  
    return c;  
}
```

# Checking Beliefs During Debugging



## Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work

If you execute the function in an [interactive debugger](#) *immediately* after writing it, e.g., set breakpoint at beginning of function and examine the values of the parameters, then single-step through code, you can manually check that your beliefs are correct for a small number of test cases

More commonly interactive debugging happens later, after tests fail, and the developer (or your future self) won't know what the beliefs were unless something present in the code tells them

They cannot know whether those beliefs were correct during previous executions unless execution logs tell them

# Code Beliefs

```
int foo (int a, int b) {  
    int c;  
    ...  
    code does something with a and b  
    ...  
    c = external_call(a,b);  
    ...  
    code does something with a, b and c  
    ...  
    return c;  
}
```

```
int foo (int a, int b) {  
    int c;  
        [ breakpoint to check a>0 and b>0 ]  
    ...  
        [ breakpoint to check a>0 and b>0 ]  
    c = external_call(a,b);  
        [ breakpoint to check a>0, b>0, c != 0 ]  
    ...  
        [ breakpoint to check a>0, b>0, c != 0 ]  
    return c;  
}
```

# Logging

Use logging framework instead of print statements!

See the [loggly ultimate guide](#) community resource covering many languages and platforms (ignore ads for loggly product)

Log entries are just messages that describes events

Severity levels, e.g., debug, info, warning, error, critical, might be configured so all levels are logged during development with only warning and above logged in production - the debug and info logging statements aren't removed, they just don't do anything

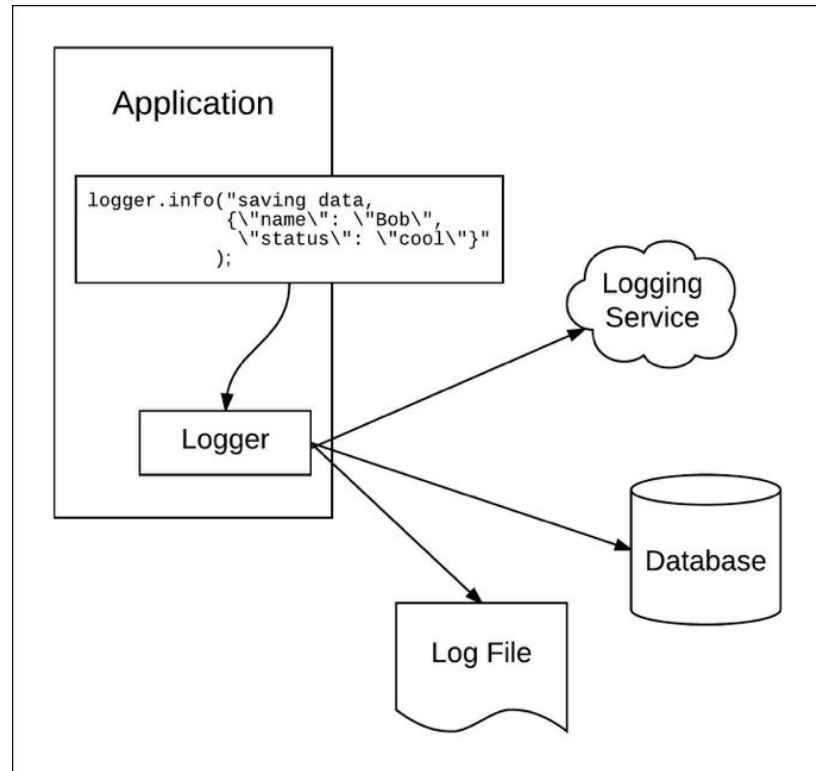
But some organizations remove log statements not intended for production before sending commits for code review

```
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
WARN LoggingUtilAgent - I am concerned...  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
INFO LoggingUtilAgent - I am happy!  
DEBUG LoggingUtilAgent - I am up, I am down, I am all araound!  
INFO LoggingUtilAgent - I am happy!  
INFO LoggingUtilAgent - I am happy!  
WARN LoggingUtilAgent - I am concerned...  
INFO LoggingUtilAgent - I am happy!  
INFO LoggingUtilAgent - I am happy!  
DEBUG LoggingUtilAgent - I am up, I am down, I am all araound!  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
WARN LoggingUtilAgent - I am concerned...  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
INFO LoggingUtilAgent - I am happy!  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
ERROR LoggingUtilAgent - I am in trouble, something went wrong.  
DEBUG LoggingUtilAgent - I am up, I am down, I am all araound!  
INFO LoggingUtilAgent - I am happy!  
DEBUG LoggingUtilAgent - I am up, I am down, I am all araound!
```



# Logging Beliefs

```
int foo (int a, int b) {  
    int c;  
    log("beginning of foo",a,b);  
    ...  
    log("middle of foo",a,b);  
    c = external_call(a,b);  
    log("after foo calls external_call",a,b,c);  
    ...  
    log("end of foo", a,b,c);  
    return c;  
}
```



# Embedded Assertions

Similar to test assertions, but embedded assertions are in the code under test, not in test

Successful assertions typically silent (nothing happens except slowing down performance)

Failed assertions typically log file name, line number, function being executed, parameters to that function, register contents, stack trace, etc. Might be configured to halt program, dump core, or jump into debugger

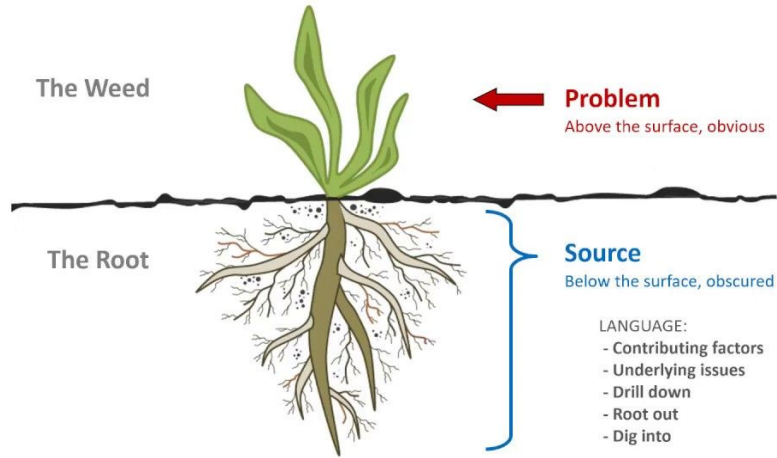
Assertions do not replace conditional error handling. Should not appear in production code except for severe violations, e.g., continuing execution would corrupt user data or system state

```
int foo (int a, int b) {  
    int c;  
    Assert(a>0 && b>0);  
    ...  
    Assert(a>0 && b>0);  
    c = external_call(a,b);  
    Assert(a>0 && b>0 && c!=0);  
    ...  
    Assert(a>0 && b>0 && c!=0);  
    return c;  
}
```

# Root Cause Analysis

Cause Mapping®

## Root Cause Analysis - The Concept



Copyright 2018 ThinkReliability

 **ThinkReliability**

Root Cause Analysis sometimes refers to treating the programming defect as a symptom, and seeking a more fundamental underlying cause in requirements, design, developer training, software process, etc.

But I just mean finding the buggy code

# Spectrum-based Fault Localization (SBFL)

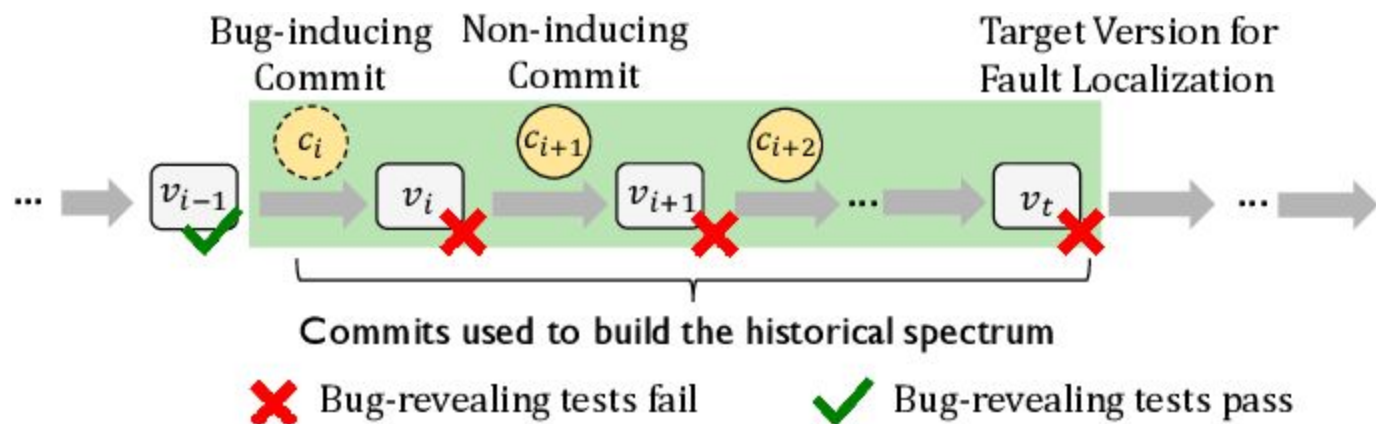
Also known as “[Round up the usual suspects](#)”

Fancy way of saying:

1. keep track of which code units had bugs in the past;
2. check which of those units, if any, are exercised by the current failing test case(s);
3. rank those units in order of which had the most previous + current bugs;
4. examine the units in order starting from the highest ranked

Works best for software with  
a substantial version history,  
but also helpful for 2nd+ iteration

Version	Initial release	Latest release
1.3	1998-06-06	2010-02-03 (1.3.42)
2.0	2002-04-06	2013-07-10 (2.0.65)
2.2	2005-12-01	2017-07-11 (2.2.34)
2.4	2012-02-21	2021-10-07 (2.4.51)



# Statistical Debugging (SD)

Similar to SBFL, but current not historical:

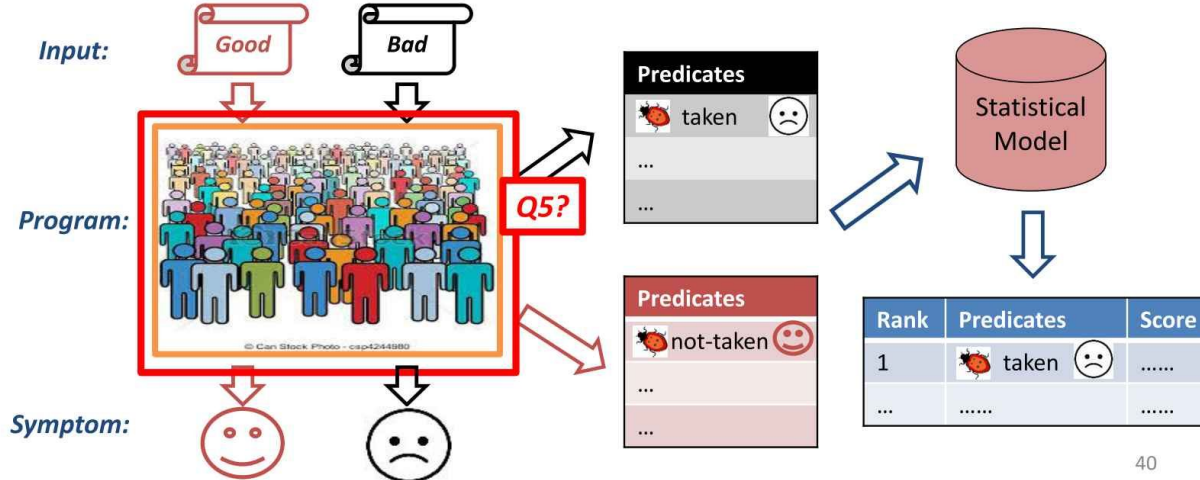
1. keep track of which parts of the code are exercised by *multiple* (current, not past) failing tests;
2. consider the parts in a hierarchy: packages, classes, methods, basic blocks, individual statements;
3. rank those parts in order of which were exercised by the most failing tests;
4. examine the parts in order starting from the highest ranked at smallest granularity

Works best when there are many failing tests, not just a few, but even with only two failing tests that exercise overlapping code - check that overlap first



# On-line Statistical Debugging

- Q5: How to do on-line performance diagnosis?
  - Less information from one single run
  - Diagnosis capability relies on multiple runs info.

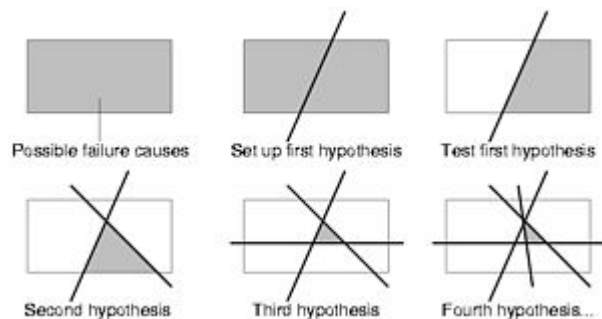


# What is the Real Difference between SBFL and SD?





# Delta Debugging



Input reduction - delta refers here to difference

Systematically remove portions of a known bug-triggering input and check whether the remainder still causes the bug. The goal is to find the smallest, simplest such input (or one of the simplest - not necessarily unique)

Most useful when initial test inputs are 'large', e.g., document or other media, data table, stream of network packets, video game save

These often come from user bug-reports or other real-world data, since the user uses real-world inputs that matter to them, not tests crafted for triggering corner cases

# Contrived Example

Initial test: buggy\_sort(4, 12, 9, 14, 3, 10, 17, 11, 8, 7, 4, 1, 6, 19, 5, 21, 2, 3)  
-> 1, 3, 2, 5, 3, 4, 4, 6, 7, 8, 9, 10, 11, 12, 14, 17, 19, 21 ❌

Reduce input to first half: buggy\_sort(4, 12, 9, 14, 3, 10, 17, 11, 8) -> 3, 12, 4, 8, 9, 10, 11, 14, 17 ❌

Reduce by half again: buggy\_sort(4, 12, 9, 14) -> 4, 9, 12, 14 ✓

Try the other half: buggy\_sort(3, 10, 17, 11, 8) -> 3, 8, 10, 11, 17 ✓

Go back to last failure-inducing input and try reducing another way - cut last two:

buggy\_sort(4, 12, 9, 14, 3, 10, 17) -> 4, 10, 9, 12, 3, 14, 17 ❌

Cutting more produces correct outputs ✓

Use the local minimum for interactive debugging (buggy code [here](#))

# Real-World Example

[This file](#) caused the Mozilla browser (now Firefox) to crash - debugging was hopeless

[This file](#), found by delta debugging, also caused the Mozilla browser (now Firefox) to crash - and drastically simplified debugging



Next Firefox Logo:



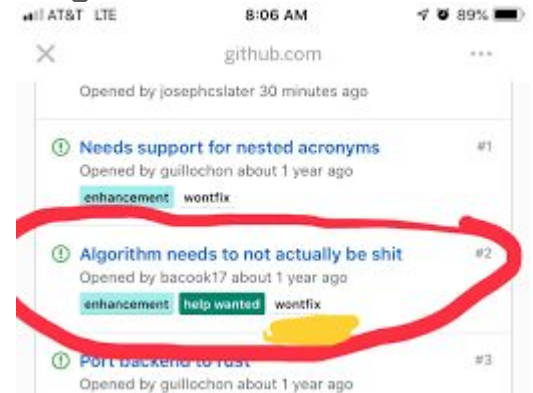
# Post-deployment Bugs

Bugs found during developer pre-deployment testing are (relatively) easy to localize - You already have a test case, likely even a unit test, that reproduces the bug!

When a user detects a bug, this usually means the developer test suite did not reveal that bug (not always, may be a case of “[wontfix](#)”)

The bug could be anywhere in the system. The developer needs to create a new test case, typically a system-level test, that reproduces and reliably detects the bug as the starting point for localizing the bug (and eventually construct a corresponding unit test)

But the user may not be able to report the step by step details to reproduce (or isn't asked to)



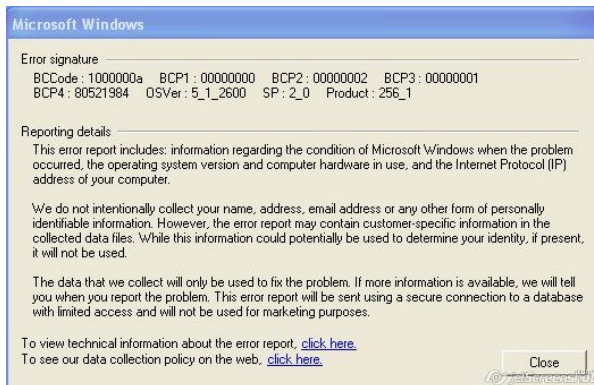
# User Bug Reports

[Bug report templates](#) make it easier for users to report bugs

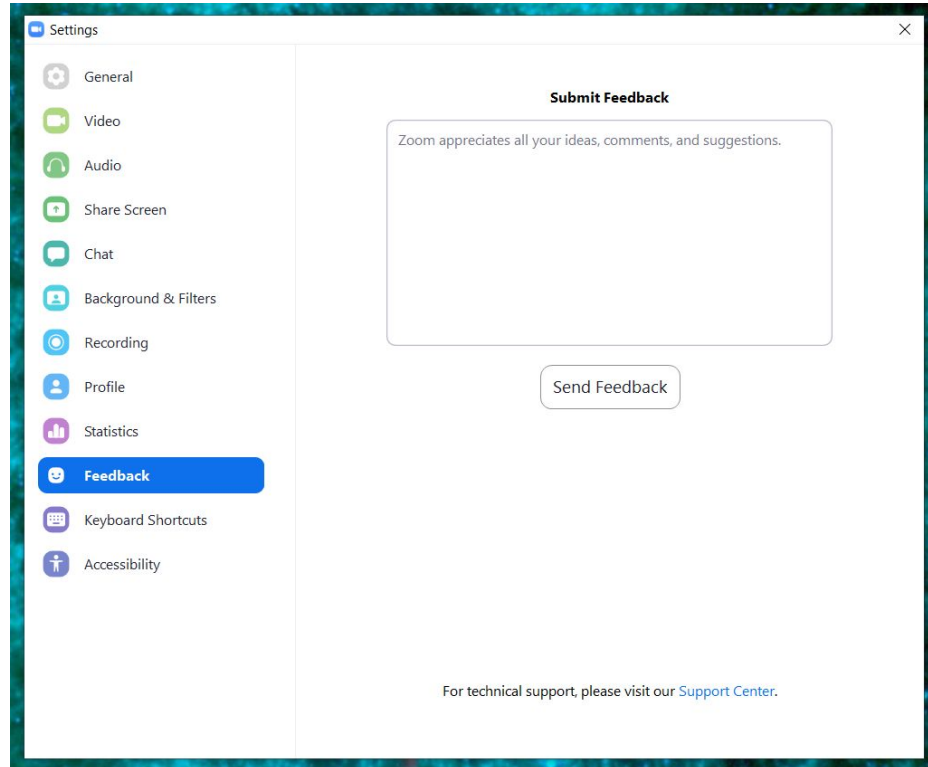
Sometimes the system provides means to include a stack trace, “core dump”, or other details automatically embedded into the bug report, rather than relying on the user’s memory of what happened

Such bug reports may support attaching a debugger “post-mortem”, so the developer can reason backwards from application state to try to construct a test case that reproduces the bug

If developers cannot reproduce the bug, the user bug report might be closed as non-reproducible (possibly reopened if other users report the same or similar failure)



# This is not a good form for reporting bugs



The image shows a screenshot of the Zoom application's 'Settings' window, specifically the 'Feedback' section. The left sidebar contains a list of settings categories: General, Video, Audio, Share Screen, Chat, Background & Filters, Recording, Profile, Statistics, Feedback (highlighted in blue), Keyboard Shortcuts, and Accessibility. The main content area is titled 'Submit Feedback' and contains a text input field with the placeholder text 'Zoom appreciates all your ideas, comments, and suggestions.' Below the input field is a 'Send Feedback' button. At the bottom of the window, there is a link to the 'Support Center' for technical support.

Settings

- General
- Video
- Audio
- Share Screen
- Chat
- Background & Filters
- Recording
- Profile
- Statistics
- Feedback**
- Keyboard Shortcuts
- Accessibility

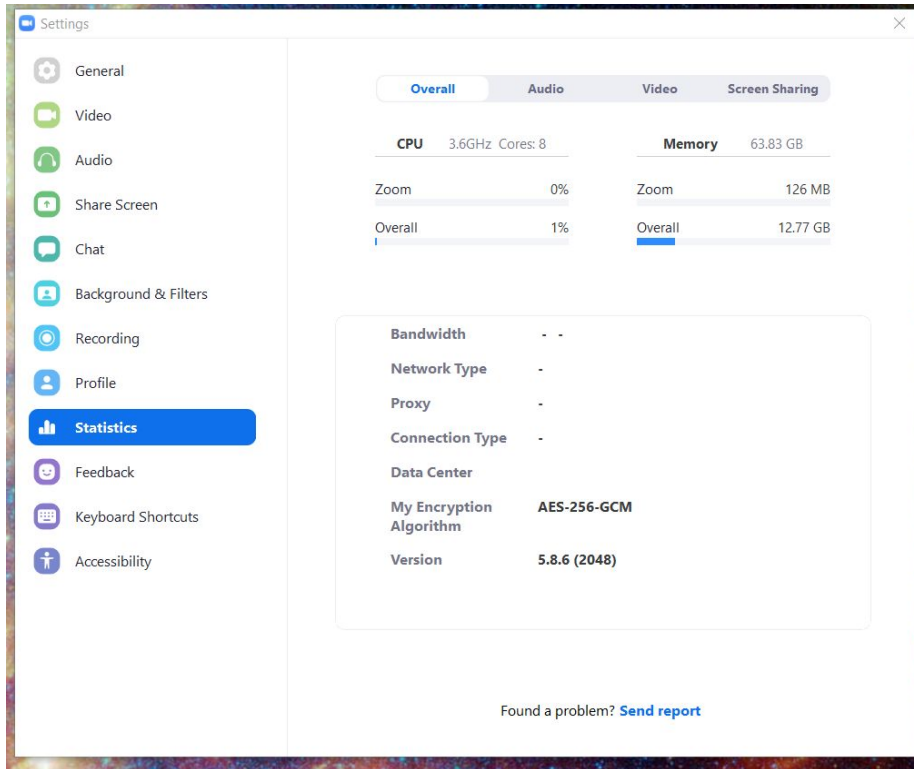
**Submit Feedback**

Zoom appreciates all your ideas, comments, and suggestions.

Send Feedback

For technical support, please visit our [Support Center](#).

# This is better



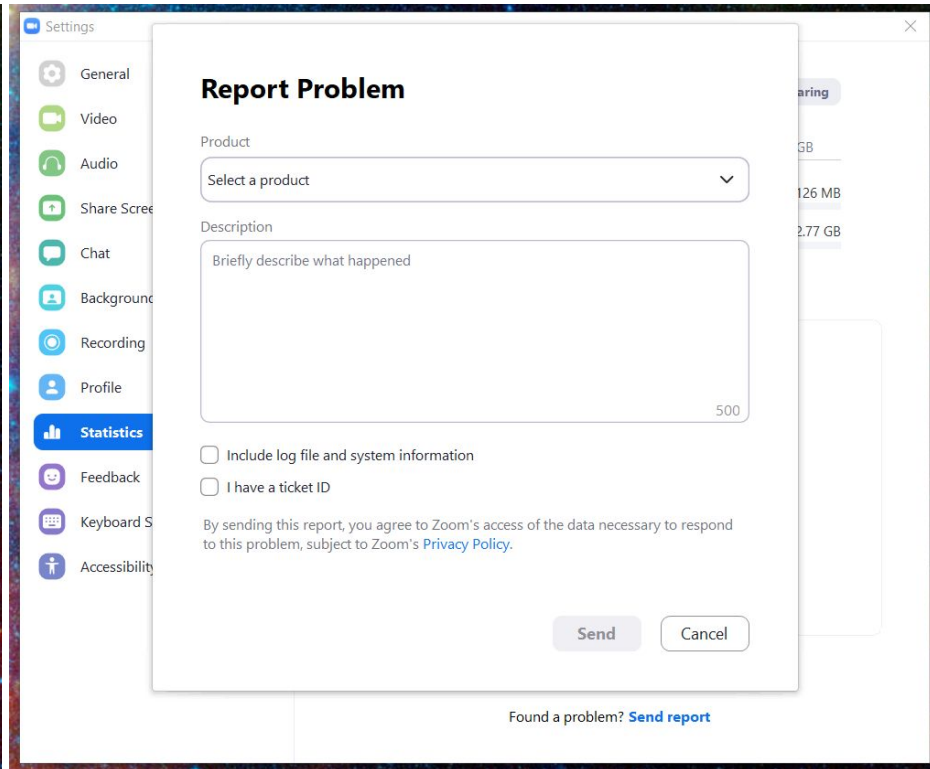
The screenshot shows the Zoom Settings application with the 'Statistics' section selected in the left sidebar. The main content area displays system performance metrics for CPU and Memory, along with network and connection details.

CPU		Memory	
Zoom	0%	Zoom	126 MB
Overall	1%	Overall	12.77 GB

Bandwidth	-
Network Type	-
Proxy	-
Connection Type	-
Data Center	
My Encryption Algorithm	AES-256-GCM
Version	5.8.6 (2048)

Found a problem? [Send report](#)



The screenshot shows the same Zoom Settings application, but with a 'Report Problem' dialog box open in the foreground. The dialog box contains a 'Product' dropdown menu, a 'Description' text area, and two checkboxes for additional information. At the bottom, there are 'Send' and 'Cancel' buttons.

**Report Problem**

Product:

Description:

☐ Include log file and system information

☐ I have a ticket ID

By sending this report, you agree to Zoom's access of the data necessary to respond to this problem, subject to Zoom's [Privacy Policy](#).

Found a problem? [Send report](#)

# Team Project

[Assignment T5: Second Iteration](#) due December 11

[Assignment T6: Second Iteration Demo](#) due December 15

[Assignment T7: Demo Day](#) on December 20