

Software Testing

CIS 3430
Week 10

Software Testing

- executing a program with the intent of finding a bug
- good tests have a high probability of finding an undiscovered error
- successful tests uncover undiscovered errors
- testing only shows that errors are present, it does not show the absence of defects

White Box testing

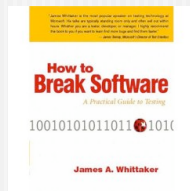
- glass box testing
- test cases built with a knowledge of the actual code
- goal is to exercise all the independent paths in the software
 - your junit testing covers this from a class perspective
- lots of other theories, we're covering none of them
 - based on graphs and complexity counts and such

Black Box Testing

- tests against the functional requirements
- finds errors in specific categories
 - interface errors
 - incorrect functionality
 - data errors
 - performance
- typically done later as components are integrated
- Again, lots of theory and opinion as to what is best
- We're looking at one, practical method

How to Break Software

James A. Whittaker



A Fault Model for Software

Users Use, Testers Test

- Any fool can stumble across bugs
- Testing requires:
 - **efficiency**: finding bugs faster than normal use
 - **effectiveness**: finding bugs that users care about and that developers will fix
 - **thoroughness**: leaving no stone unturned

Where are the Bugs?

- To find them you could:
 - Seek out the weak developers!
 - Seek out the bad managers!
 - Seek out the doomed projects!
 - Create them yourself!
- Understanding where bugs are requires that we understand *how and why software fails*

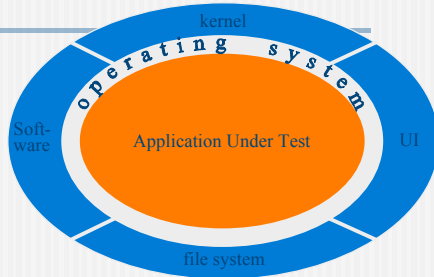
Software Fault Models

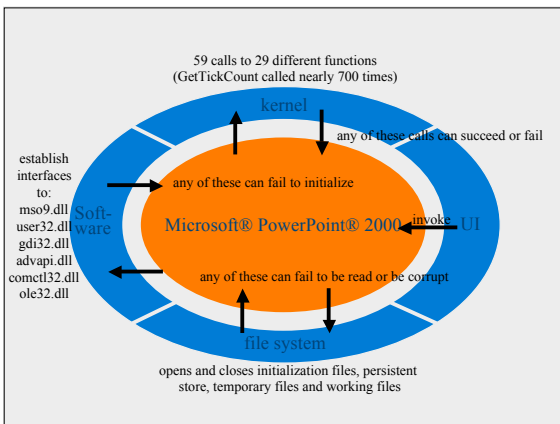
- Fault models can be based on:
 - Process maturity
 - Programming language constructs
 - Software behavior
- How do we understand problematic behavior?
 - Read bug reports
 - Recognize patterns of failure

Where Bugs Come From

- Environment
 - The software misinterprets or cannot handle its environment
 - What are all the environmental considerations that we must face?
- Capabilities
 - The software incorrectly performs one or more of its capabilities
 - What are software's general capabilities?

Understanding the Environment





Environment Interaction is Complex!

Let's take a closer look
at each of these
interfaces

The Human Interface

- Two main types of human interfaces:
The GUI: inputs arrive through GUI controls from the mouse or keyboard

1

some controls have only simple events
some controls can deliver data which must be validated
some controls affect other controls
the order in which controls are exercised is an important consideration

The Human Interface

Two main types of human interfaces:

The GUI: inputs arrive through GUI controls from the mouse or keyboard

1

• How human inputs actually get transmitted to software is a point of interest

The Human Interface

- Two main types of human interfaces:

The API: inputs arrive from programs
some parameters are simple
some parameters contain or point to data
parameter combinations have to be considered
the sequence in which specific API calls are made is an important consideration

2

The Kernel Interface

- The operating system user is very complex
 - The Windows kernel exports over 1000 functions
 - Each function has at least two return values
 - Too often, developers trust the kernel explicitly

The Kernel Interface

- How do we test the kernel interface?
 - Run a bunch of background apps to:
 - slow performance
 - simulate resource contention
 - test low memory situations
 - But...
 - How do we repro these bugs?
 - How can we debug the app?

The Kernel Interface

- The fundamental problem is that the kernel interface is *reactive*
- We need some good system tools to expose this interface to testers and give us the ability to *proactively* determine what return values our app sees

The Software Interface

- Examples:
 - Making SQL queries to an external database
 - Using a socket API to manage a network connection
 - Using a math library
 - Using third party GUI controls

The Software Interface

- Fact: Run time libraries and reused components are SOFTWARE
- Fact: SOFTWARE fails!
- Fact: Since SOFTWARE fails, we cannot trust software users

The Software Interface

- Software users can:
 - Work properly (be reliable)
 - Pass unexpected return values
 - Pass bad data
 - type
 - format
 - Crash (GP fault, core dump)

The File System Interface

- Files are users, their contents are inputs
- File contents can:
 - be perfect
 - have the wrong permissions
 - get deleted by another user
 - be corrupt
 - bad data
 - misplaced delimiters

Software Capabilities

Now that we understand software environments, lets take a close look at software capabilities

Testing is War

- The enemy: bugs in the software
- Bugs prevent capabilities
- Drive capabilities and you find bugs
- Method:
 - Determine your enemy's strengths and remove them
 - Wage small wars on input capabilities, output capabilities, data storage and computation

Testing Input

- Software should only accept input that it can handle...

... But ensuring that this is the case is problematic

Testing Input

- How does software filter erroneous input?
 - GUIs
 - 1 • by preventing input data of incorrect type
 - by preventing input data that is too small or too large
 - by forcing the user down specific control flow paths

Testing Input

- How does software filter erroneous input?
 - Error checking code
 - “if” statements
 - 2** • by ensuring that inputs received can actually be processed
 - but error checking code can also have errors!
 - writing error code might introduce errors
 - writing error code also means diverting your attention from the main-line code

Testing Input

- How does software filter erroneous input?
 - Exception handlers
 - failing gracefully is extremely difficult
 - 3** • error routines must reset state and cleanup side-effects... a very difficult endeavor

Testing Output

- Software should generate only those outputs that are acceptable to its users
 - Displayed data must fit in its display area
 - Software cannot pass incorrect data types
 - Data must be correctly computed, software must *never* pass incorrect values to its users
 - Testing output requires domain expertise
- We must understand wrong answers and ensure that our software does not produce them

Testing Data

- Inputs and computation results are often stored internally
- Software will fail if it stores illegal data
- Stored data values must be acceptable individually and in combination with other data

Testing Data

- The major difference between data and inputs is that data is *persistent*
- Persistence needs to be tested
 - data retrieval, data modification, data access
 - we must test that data structures can be operated on without failure
 - accessed, retrieved, modified, overflowed, underflowed, ...

Testing Computation

- Software can correctly filter inputs, validate outputs and store data...
... *and still fail*
- $x=x+1$ will fail if it is executed enough times to overflow the value x
- Correct computation depends on operators, operands *and* result

Testing Computation

- Another aspect of computation is feature interaction
 - Features can interact in ways that affect computation
 - One feature can *get in the way* of another feature's computation

Software Testing

- Testing consists of:
 - Establishing an environment in which to execute the system under test
- This means controlling:
 - The user interface
 - The operating system kernel
 - Other software systems
 - The file system

Software Testing

- Testing consists of:
 - Driving the system to exhibit its capabilities
- This means exercising the software's:
 - Input domain
 - Output range
 - Internally stored data
 - Computation

How to Break Software

Act 1: The User Interface

1

An Overview of the Methodology

- Software possesses 4 basic capabilities
- Attack each capability by staging situations that commonly cause failure
 - input attacks
 - output attacks
 - data attacks
 - computation attacks
- Determine which attacks apply to your app and apply them, one-at-a-time

Exploring the Input Domain

- Banging on the keyboard is largely a waste of time, a strategy for rookies
- Each test should have a specific *purpose*
- A tester with clear goals is more likely to find a problem than a tester who is simply hacking away
- Testers must learn to target problematic input scenarios

Exploring Inputs

1

- Explore the application under test
 - Apply the inputs that a user would apply to get real work done
- Observe the inputs
- Watch for attack opportunities

Preview of Input Attacks

1. Force all error messages to occur
2. Force the software to establish default values
3. Explore allowable character sets and data types
4. Overflow input buffers
5. Find inputs that interact with other inputs
6. Repeatedly apply the same input/input sequence

Documenting an Attack

- When to apply the attack
- What faults make the attack successful
- How to determine if the attack exposes failures
- How to conduct the attack
- Example and analysis

First Attack: Force Error Messages to Occur

- When to apply this attack:
 - Whenever an application must respond to erroneous input
 - Testers should consider the type of response:
 - Input filter
 - Input checking
 - Exception handling

Attack 1

Force Error Messages to Occur

- What faults make this attack successful:
 - Programming error cases is difficult
 - Additional code must be written—where there is code, there is often bad code
 - Writing error code takes the programmers attention away from writing functional code
 - Failing gracefully is difficult
 - What data needs to be saved?
 - What state is the app in?

Attack 1

Force Error Messages to Occur

- How to determine if the software fails:
 - This attacks finds
 - Missing error cases
 - Nonsense error messages
 - Uninformative error messages
 - Thus, manifestation of the defect ranges from crash to fully functional code

Attack 1

Force Error Messages to Occur

- How to conduct this attack:
 - Look thru the (ahem) specification for message definition
 - Consider *properties* of inputs
 - [type] entering invalid types often causes error messages
 - [length] a few too many characters in an input string will often elicit an error message
 - [boundary values] often represent special cases
 - index example in MS Word
 - if time the gallery of error messages

Attack 1

Second Attack: Force Default Values

- When to apply this attack:
 - All software uses variables
 - The lifecycle of a variable is
 - Declaration
 - Initialization
 - Use
 - Variables must be initialized before they are used

Attack 2

Force Default Values

- What faults make this attack successful:
 - Compilers are often good at catching these mistakes
 - But the compilers must be used properly
 - Implicit declaration can often get a programmer into trouble
 - When users skip input fields or leave them blank, defaults must be established in case those variables are used in computation

Attack 2

Force Default Values

- How to determine if the software fails:
 - Best case (for ease of verification) is that use of an un-initialized variable causes a memory violation
 - Harder to detect is that a random value gets assigned to the variable
 - Look for garbage characters/"strange things"
 - Incorrect results
 - Too many values/too few values displayed
 - Incorrectly typed data being displayed

Attack 2

Force Default Values

- How to conduct this attack:
 - Determine the data that has defaults
 - Look for "options" or "configuration" buttons
 - Consult the source code's declaration section
 - Force the app to use its defaults
 - Accept any displayed data as defaults
 - Enter a null value, if a value is displayed then delete it
 - Change settings from their default values to a valid value and then back again
 - MS Word, table of contents example

Attack 2

Third Attack: Explore Character Sets/Data Types

- When to apply this attack:
 - Target: variable input
 - Special cases based on:
 - Operating system reserved words
 - Programming language reserved words
 - Character set boundaries (ASCII, UNICODE, etc.)
 - spaces, quotation marks, delimiters, etc.

Attack 3

Explore Character Sets/Data Types

- What faults make this attack successful:
 - Special cases require special handling
 - Either:
 - Developers fail to recognize a special case
 - Developers put too much trust in interface controls
 - Developers fail to handle errors properly (we've already discussed that error code is hard to get right)

Attack 3

Explore Character Sets/Data Types

- How to determine if the software fails:
 - Unhandled exceptions cause a system to crash
 - Generalized error handlers often present nonsensical or uninformative messages
 - Watch out for loss-of-state often caused by exception handlers
 - Since we are dealing with character data, watch for rendering problems
 - Watch for "Easter eggs"

Attack 3

Explore Character Sets/Data Types

- How to conduct this attack:
 - Spend some time researching:
 - Operating system, programming language and character set keywords and ranges
 - Study documentation of each of the above
 - Beware outdated keywords

Attack 3

Fourth Attack: Overflow Input Buffers

- When to apply this attack:
 - The idea is simple: enter long strings into input fields
 - Don't neglect APIs/exposed internal objects
 - This is the hacker's choice because many buffer overflows create exploitable failure scenarios

Attack 4

Fourth Attack: Overflow Input Buffers

- When to apply this attack:
 - This is an important bug because:
 - copy/paste into inputs fields is a fairly common practice
 - Buffer overruns result in crashes, risking data loss and costly rework
 - be sure to check with developers to find out tolerance for fixing long string bugs...thousands characters can get ridiculous

Attack 4

Overflow Input Buffers

- What faults make this attack successful:
 - Developers simply fail to constrain the amount of text the software will accept in an input sting
 - When the text is read input memory, fixed-sized buffers are overflowed

Attack 4

Overflow Input Buffers

- How to determine if the software fails:
 - This bug almost always causes the software to crash
 - Other possibilities are extreme application instability (since memory is often overwritten by the long string)

Attack 4

Overflow Input Buffers

- How to conduct this attack:
 - Identify where strings are read as input
 - Start small and then grow the string to its maximum length
 - This attack is repetitive
 - It's helpful to count 12345678901234567890
 - don't go overboard- try lengths that could really occur

Attack 4

Fifth Attack: Find Interacting Inputs

- Up to now, we have dealt with inputs one-at-a-time
 - Find a spot where input is accepted and poke it until something breaks
- This next attack deals with *combinations* of inputs
 - Multiple inputs on a single input dialog
 - An API call with more than one input
- Constraining a single input is hard enough for developers...

Attack 5

Fifth Attack: Find Interacting Inputs

- When to apply this attack:
 - Some inputs affect other inputs
 - They might represent different properties of the same piece of data
 - They might be used in a single internal computation
 - Thus, individually correct inputs might be problematic when combined
 - These relationships should be watched for while executing the previous attacks

Attack 5

Find Interacting Inputs

- What faults make this attack successful:
 - Obviously, input relationships are hard to determine for both testers and developers
 - The logic involved in handling a single erroneous input is hard enough...
 - ...multiple error cases often require complex nested IF statements
 - Code changes make this situation worse

Attack 5

Find Interacting Inputs

- How to determine if the software fails:
 - Since inputs are often stored internally, slipping a bad input in means corruption of internal data
 - Once you suspect you tricked the app into accepting bad input, force that input to be used as much as possible
 - Carefully verify it every time it is displayed or used

Attack 5

Find Interacting Inputs

- How to conduct this attack:
 - Explore the app, identify possible relationships between inputs
 - Are they properties of a single data structure?
 - Are they used together in a single computation?
 - Select boundary and extreme combinations:
 - Large/small, small/large, large/large, small/small

Attack 5

Sixth Attack: Repeat Inputs Numerous Times

- When to apply this attack:
 - This attack is applicable whenever the app accepts input inside a loop:

Accept an input
Process it
Repeat

Attack 6

Repeat Inputs Numerous Times

- What faults make this attack successful:
 - Repetition has the effect of gobbling up resources
 - Many applications are unaware of available resources and assume unlimited memory and storage
 - Low resources can cause undesirable side-effects

Attack 6

Repeat Inputs Numerous Times

- How to determine if the software fails:
 - It is difficult to predict how memory stress will manifest
 - Watch for:
 - Screen refresh problems
 - Slow performance
 - Consider:
 - Using a memory leak detector

Attack 6

Repeat Inputs Numerous Times

- How to conduct this attack:
 - Pick an input or a sequence of inputs
 - Apply them over and over to test for undesirable side-effects
- Or:
 - Pick an object and apply the same input to it over and over
 - Pick multiple objects and apply the same series of inputs to each object

Attack 6

Review of Input Attacks

1. Force all error messages to occur
2. Force the software to establish default values
3. Explore allowable character sets and data types
4. Overflow input buffers
5. Find inputs that interact with other inputs
6. Repeatedly apply the same input/input sequence

Exploring Outputs

2

- Some bugs are too difficult to find by concentrating on inputs alone
- Which inputs will generate incorrect results?
- Why not concentrate on what incorrect results *could* occur and then find the inputs to force them?

Preview of Output Attacks

7. Force different outputs to be generated for each input
8. Force invalid outputs to be generated
9. Force output properties to change
10. Force the screen to be refreshed

Seventh Attack: Vary Outputs for Each Input

- When to apply this attack:
 - Inputs are not independent of each other
 - Applying some inputs before (or after) other inputs can affect behavior
 - A good way to think about the most important cases is to ensure that you see every output that an input can cause

Attack 7

Vary Outputs for Each Input

- What faults make this attack successful:
 - Input → Output is simple to code
 - Inputs that cause different outputs require complex logic to be coded
 - Complexity leads to bugs

Attack 7

Vary Outputs for Each Input

- How to determine if the software fails:
 - The main bug is that there are behaviors that the developer miscoded or forgot to code
 - These are usually severity 1 problems that get found and fixed before release

Attack 7

Vary Outputs for Each Input

- How to conduct this attack:
 - Testers must identify which inputs can cause multiple behaviors and understand the *context* in which these inputs can be applied

Attack 7

Vary Outputs for Each Input

- To illustrate, consider a telephone switch
- How many outputs can the input “pick up the receiver” cause?
 - The switch could **generate a dial tone** ...
... when the phone is idle
 - The switch could **connect two callers**...
... when the phone is ringing

these
are the
situations
to test!

Attack 7: Example

Eighth Attack: Force Invalid Outputs

- When to apply this attack:
 - Domain expertise is essential to effectively carry out this next attack
 - It's hard to test a flight simulator without knowing how to fly an airplane
 - Testers must know the product well enough to enumerate wrong answers...
 - ...and then figure out how to drive the application to produce them

Attack 8

Force Invalid Outputs

- What faults make this attack successful:
 - Just as testers can misunderstand the problem domain...so can developers
 - When this happens, they write bugs
 - In most cases, the problem is over-looked special cases

Attack 8

Force Invalid Outputs

- How to determine if the software fails:
 - This is a hard one...the software rarely fails in a spectacular (or even noticeable) manner
 - Testers should ignore *type* and *format* and concentrate instead on the *value* being displayed

Attack 8

Force Invalid Outputs

- How to conduct this attack:
 - Testers need to focus on *known bad results*
 - Enumerate wrong answers and then create the situation that forces them
 - Learn as much about the problem domain as you can

Attack 8

Ninth Attack: Change Output Properties

- When to apply this attack:
 - This attack requires *persistent* outputs be present in the application (which often precludes its use on APIs)
 - An output is generated and then we change some property of the output

Attack 9

Change Output Properties

- What faults make this attack successful:
 - Generating the output once tests that the software works with initial data settings
 - These are settings that the developer established
 - These are settings that the developer has anticipated
 - Changing the output ensures that the software will work user-defined setting
 - These settings might not be anticipated

Attack 9

Change Output Properties

- How to determine if the software fails:
 - Since outputs are necessarily visually-intensive, this often requires manual verification
 - But since testers have had to determine output properties in advance, it is easy to determine what to look for

Attack 9

Change Output Properties

- How to conduct this attack:
 - First determine the property of interest
 - Size, value, type, color, shape, direction, ...
 - Cause it to be displayed, then change it
 - Size (bigger to smaller, smaller to bigger,...)
 - Value(s) (high to low, positive to negative,...)
 - Type (char to int, int to float,...)
 - ...

Attack 9

Tenth Attack: Force the Screen to Refresh

- When to apply this attack:
 - This attack is applicable only to GUI software with editable display areas
 - The attack is most useful at the boundaries of screen objects
 - Objects created by the user
 - Partitions of the display area (frames, etc.) set by the software

Attack 10

Force the Screen to Refresh

- What faults make this attack successful:
 - Refreshing the contents of a window, after those contents have changed, is problematic
 - Refresh too often:
 - Performance degrades
 - The screen flickers to annoy the user
 - Refresh too seldom
 - The screen becomes messy
 - Hitting the sweet spot is challenging

Attack 10

Force the Screen to Refresh

- How to determine if the software fails:
 - Sigh...you guessed it...labor-intensive visual verification is the only resort

Attack 10

Force the Screen to Refresh

- How to conduct this attack:
 - Add things to the screen
 - Move them around (varying the distance you move them)
 - Delete them
 - Edit them
- Do all these things with a mix of features and at or around object boundaries

Attack 10

Review of Output Attacks

7. Force different outputs to be generated for each input
8. Force invalid outputs to be generated
9. Force output properties to change
10. Force the screen to be refreshed

Exploring Stored Data

3

- While executing the input and output attacks, keep notes on persistent (stored) data
 - Data you see over and over is stored
 - Data you see over multiple executions is persistent
- Think about where this data comes from and how it can get corrupted

Exploring Stored Data

- If you have the source code, data is easy to find
- If you don't have the source, you must be able to look through the interface and identify data
 - Put yourself in the shoes of the developer
 - How would you program it?

Preview of Stored Data Attacks

11. Apply inputs using a variety of initial conditions
12. Force a data structure to store too many/too few values
13. Investigate alternate ways to modify internal data constraints

Eleventh Attack: Vary Initial Conditions

- When to apply this attack:
 - Inputs are often applicable in a variety of circumstances (states of the software)
 - This attack investigates the application of inputs from a variety of initial states
 - Try to pick states that might cause errant behavior
 - Pick states that are as different from each other as possible
 - Pick states that differ by only one data element

Attack 11

Vary Initial Conditions

- What faults make this attack successful:
 - Checking that inputs are valid is tricky
 - Checking that inputs—combined with internal state—are valid is even harder
 - The result is that some inputs only work on specific initial conditions

Attack 11

Vary Initial Conditions

- How to determine if the software fails:
 - Since some cases of this bug mean missing code (failure to consider a specific case)
 - Watch for crashes
 - Watch for nonsense error messages

Attack 11

Vary Initial Conditions

- How to conduct this attack:
 - Enumerating states is a time consuming endeavor:
 - Does an input cause different behavior in different circumstances?
 - Is an input applicable sometimes but not other times?

For more information read about *model-based testing*
visit: www.model-based-testing.org

Attack 11

Twelfth Attack: Over/Underflow Data Structures

- When to apply this attack:
 - Fixed size structures can be overflowed by forcing the software to store too many values
 - Testers must be able to detect size constraints and then find ways to violate them

Attack 12

Over/Underflow Data Structures

- What faults make this attack successful:
 - Developers are often good at checking content and failing to consider size
 - Every program that adds or removes elements from a structure must have code to check that size constraints are not broken as a result of the add/remove operation
 - It only takes one program to forget...

Attack 12

Over/Underflow Data Structures

- How to determine if the software fails:
 - Reading or writing beyond the bounds of a data structures almost always causes the software to crash
 - But be alert for the usual signs of data corruption (performance, anomalous output/computation, etc.)

Attack 12

Over/Underflow Data Structures

- How to conduct this attack:
 - A structure can be underflowed by removing more values than the structure contains
 - Strategy:
 - add x number of values to a structure
 - delete $x+1$ values from the structure
 - Try this on internal structures and also list boxes that store persistent data

Attack 12

Find Back Doors to Stored Data

- When to apply this attack:
 - Whenever data is supposed to be constrained in any way:
 - A month must fall between 1 and 12
 - A year must fall between 1980 and 2095
 - The limit on undo operations is 20
 - ...

Attack 13

Find Back Doors to Stored Data

- What faults make this attack successful:
 - Plain and simple: a lack of good software engineering practice
 - Information hiding (object-orientation) cures this problem:
 - All data is private to a set of access routines
 - These access routines enforce data constraints
 - A program must use the access routines to access or modify data

Attack 13

Find Back Doors to Stored Data

- How to determine if the software fails:
 - Broken data constraints are serious, often resulting in system instability or crash
 - Also, look for:
 - System sluggishness
 - Incorrect error messages
 - Incorrectly computed results

Attack 13

Find Back Doors to Stored Data

- How to conduct this attack:
 - Identify data structures
 - Explore various ways to modify the contents or properties of the data structure

Attack 13

Review of Stored Data Attacks

11. Apply inputs using a variety of initial conditions
12. Force a data structure to store too many/too few values
13. Investigate alternate ways to modify internal data constraints

4

Exploring Computation

- While executing the input and output attacks:
 - make a feature list
 - keep notes on what computation is happening inside the application
- Think about ways to “get in the way” of the computation, using inputs, outputs, data, or even other features

Preview of Computation Attacks

14. Experiment with invalid operand and operator combinations
15. Exploit recursion
16. Force computation results to be too large or too small
17. Find features that share data or interact poorly

Fourteenth Attack: Experiment w/ Operator/Operands

- When to apply this attack:
 - Some operands cannot be used with certain operators
 - Divide by zero
 - Adding a character to a real number
 - Some operators conflict with each other
 - Operators with an inverse relationship
 - Ex: $[\text{SQRT}(x)]^2$

Attack 14

Experiment w/ Operator/Operands

- What faults make this attack successful:
 - Developers have to write error code to weed out invalid operator/operand combinations
 - Consider divide-by-zero

```
if (x != 0)
  y = z / x;
else
  cout("cannot divide by zero");
```

Experiment w/ Operator/Operands

- How to determine if the software fails:
 - Become a problem domain expert to learn about the domain and range of the functions computed by your software
 - Apply invalid combinations and watch for:
 - Crashes (for unhandled exceptions)
 - Incorrect results
 - Null or nonsensical values appearing as output

Attack 14

Experiment w/ Operator/Operands

- How to conduct this attack:
 - Must be able to identify computation (operators) and the data objects (operands) on which the computation operates
 - Computation isn't just assignments
 - Graphical rendering
 - Screen layout, WYSIWYG

Attack 14

Fifteenth Attack: Exploit Recursion

- When to apply this attack:

- Recursion occurs when a function calls itself

```
long int factorial(long int n)
{
    if (n <= 1)
        return(1);
    else
        return(n * factorial(n - 1));
}
```

- If this occurs too many times, stack overflow will occur

Attack 15

Exploit Recursion

- What faults make this attack successful:

- Developers fail to write code that ensures loop or recursive call termination
- Since recursion only requires one line of code, constraining it often seems unnecessary to developers

Attack 15

Exploit Recursion

- How to determine if the software fails:

- Infinite recursion causes a heap overflow in the computer's memory
- This will almost always result in the application crashing or hanging

Attack 15

Exploit Recursion

- How to conduct this attack:
 - Recursion can be thought about in terms of black box objects too:
 - Can a document reference itself?
 - Can a hyperlink point to itself?
 - Can an email message contain itself?
 - Can a program that spawns processes call itself?
 - Determine these objects and create the self-referencing situation

Attack 15

Sixteenth Attack: Force Results to be too Large/Small

- When to apply this attack:
 - Review:
 - Inputs need to be constrained
 - an app should never permit invalid input to enter the system
 - Data needs to be constrained
 - an app should never store invalid data
 - But even if all inputs and data are correct, computation can still fail

Attack 16

Force Results to be too Large/Small

- What faults make this attack successful:

```
x=x+1;
```

- This simple line of code will fail if it is executed enough times
- Developers often focus on operators and operands and ignore the result

Attack 16

Force Results to be too Large/Small

- How to determine if the software fails:
 - Since the result of this attack is underflow or overflow, the software most often crashes

Attack 16

Force Results to be too Large/Small

- How to conduct this attack:
 - Usually, you have to force a computation to occur over and over and over
 - If you have control over the data used in the computation, rig it to begin at or around boundary values

Attack 16

Force Results to be too Large/Small

- Given the program:

```
#define count 2
main() {
    int sum, value[count];
    sum = 0;
    for (i = 0; i < count; ++i) {
        sum = sum + value[i];
    }
}
```
- Consider the values:
 - value[0] = 32700, value[1] = 70
 - count = 33000, value[0..32999] = 1

Attack 16: Example

Seventeenth Attack: Feature Interaction

- When to apply this attack:
 - This is the attack that distinguishes novice testers from the pros
 - Given two or more features that work fine individually, can you make them break when they work together
 - on the same data
 - at the same time
 - getting in the way of each other

Attack 17

Feature Interaction

- What faults make this attack successful:
 - Features often work well in isolation because they are developed in isolation
 - But two features may share data but require that different constraints on that data be enforced
 - Failure to enforce such constraints causes this class of failure

Attack 17

Feature Interaction

- How to determine if the software fails:
 - Here is yet another example of painstaking manual verification
 - Keep a sharp eye out for:
 - Improperly formatted output
 - Incorrect computation
 - Corrupted data
 - Failures often manifest long after the fault has been tripped over

Attack 17

Feature Interaction

- How to conduct this attack:
 - Pick two or more features
 - That might interact with the same user input
 - That might contribute to computing a single output
 - That might share the same data
 - Try to make them “get in each other’s way”

Attack 17

Review of Computation Attacks

14. Experiment with invalid operand and operator combinations
15. Exploit recursion
16. Force computation results to be too large or too small
17. Find features that share data or interact poorly

How to Break Software

Act 2: The Kernel Interface

2

The Kernel Interface

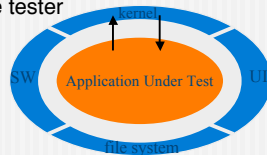
- Although the kernel is responsible for numerous application services, it's most central task is memory management
- How can memory cause applications to fail?
 - The system can run out of memory
 - Apps can contend for resources
 - The operating system can fail

Testing the Kernel Interface

- Testing OS and memory failures is problematic:
 - Running dozens of background apps is time-consuming and requires significant monitoring and cleanup
 - Debugging root cause is guess work...which calls are actually failing?

Kernel Imposters

- Holodeck uses a kernel imposter that:
 - logs all app-to-kernel activity
 - allows kernel-to-app return values to be controlled by the tester



How to Effectively Fault Inject the Kernel Interface...

- ...Is an open question
- Certainly:
 - It is environment specific
 - handhelds, embedded systems, desktop apps all have different thresholds of acceptable behavior
 - It is application specific
 - some apps must be fault tolerant or provide a high degree of safety or security
 - some apps can fail and nobody cares, including the developers

Fault Injection Strategy

- There are two approaches:
 - Monitor your application in a stressed environment
 - 1** Use Holodeck to record the traffic across the kernel interface
 - Simulate a similar pattern of failed calls for testing successive builds of the same product

Fault Injection Strategy

- There are two approaches:
 - Systematically fail each call and document the application's behavior
 - 2** Get with developers to analyze root cause and decide remedies (if any)
 - Thoroughly test each successive build for problematic calls

How to Break Software

Act 3: The File System Interface

3

The File System Interface

- Nearly all applications use the file system
- How can the file system cause applications to fail?
- Categories of potential problems:
 - Media problems
 - File problems

Media-based Attacks

- Fill the file system to its capacity
- Force the media to be busy or unavailable
- Damage the media

File-based Attacks

- Assign an invalid file name
 - Too long
 - Invalid character set/format
- Vary access permissions
 - Read only, user privileges
- Vary/corrupt file contents

How to Break Software

Act 4: The Software Interface

4

The Software Interface

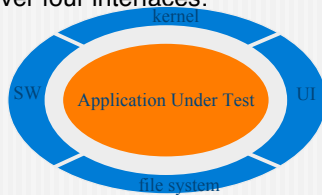
- Most complex applications are built from components, many of them reused
- How can reused software cause our application to fail?
 - Reused software is software...we should expect it to fail!
 - It can pass unexpected return values
 - It can pass unexpected or corrupt data

Fault Injecting the Software Interface

- Same two approaches for the kernel interface apply also to software:
 - Monitor the calls...inject realistic faults...simulate by replaying the interesting scenarios
 - Systematically fail each call

Summary

- Software testing requires understanding of and control over four interfaces:
 - UI
 - Kernel
 - File System
 - Software



Summary

- Each interface presents number of challenging problems
- Testing any of the interfaces is formidable
- Testing them thoroughly is exquisitely difficult
- Some interfaces are well-understood, some we are just beginning to study

Conclusions

- Testing each interface:
 - The user interface
- 1**
- study the attacks
 - execute the attacks
 - meet often, share best practices

Conclusions

- Testing each interface:
 - The kernel interface
- 2**
- acquire the tools
 - acquire the knowledge
 - study field bug reports and make sure you understand real stressed environments

Conclusions

- Testing each interface:
 - The file system interface
- 3**
- understand your app's file formats
 - understand how storage media can fail and determine your customer's tolerance for losing their data

Conclusions

- Testing each interface:
 - The software interface
 - 4 • understand the other apps that your software depends upon
 - understand the data that your app gets from these resources
 - determine how this communication can fail
 - determine if failure of the resource will cause failure of your app

Remember

- Know your app's environment
 - Understand what might go wrong and test that it doesn't
- Know your app's capabilities
 - Understand how it can screw things up, test that it doesn't
- Practice the attacks... *always* have a goal
- Brain on, eyes open, Test!

Configuration and Installation

- Once you have it written and tested, then what?
