Presentation
Paper
Bio

Return to Main Menu

P R E S E N T A T I O N

WG2

Wednesday, Dec 6, 2000

How to Break Software (with examples)

Alan Jorgensen

International Conference On
Software Testing, Analysis & Review
DEC 4-8, 2000 • COPENHAGEN, DENMARK

How To Break Software

Alan A. Jorgensen

Advanced Engineering Technology, Inc.
1900 S. Harbor City Blvd., Suite 115
Melbourne, Florida 32901
aaj@aet-usa.com

James A. Whittaker

Florida Tech—Computer Science
150 West University Boulevard
Melbourne, Florida 32901
jw@cs.fit.edu

Presented by Dr. Alan A. Jorgensen, Ph. D.

Users Use, Testers Test

AET, Inc
Advanced Engineering Technology

- Any fool can stumble across bugs
- Using software hoping to find bugs is ineffective
- Being a tester means finding bugs efficiently
- How do we do this?
  - Set clear goals for every test case
  - Understand where bugs might hide
  - Know how to expose them

How to Break Software

AET, Inc
Advanced Engineering Technology

- Our method:
  - Collect and study a large number of bugs in released products
  - Understand why they occur and what type of tests it would take to find them
  - Generalize the tests into “attack patterns” and teach students how to execute these patterns
  - Collect even more bugs, classify them and refine the attacks

Exploratory Testing

AET, Inc
Advanced Engineering Technology

- **Main Testing Problem**
  - Coded without clear specifications.
  - We must test a product whose purpose is unclear
- **Learn the Application Domain**
- **Explore the Software by Plan**
- **Utilize a Failure Model**

Failure Model

AET, Inc
Advanced Engineering Technology

- **Software Fails because Developers Fail to Properly Constrain:**
  - Inputs
  - Outputs
  - Storage
  - Computation
- Surprised? This is what software does:
  - Input, store, compute, and output
- *cf*: STAREAST 1999, “Why Software Fails”

Testing Strategy

AET, Inc
Advanced Engineering Technology

- **Study the inputs**
  - Run attacks to break input constraints
- **Study the outputs**
  - Run attacks on output constraints
- **Study the way software stores data**
  - Run attacks designed to corrupt internal data
- **Study how the software does computation**
  - Run attacks to force errant computation

Input Constraint Attacks

AET, Inc
Advanced Engineering Technology

- **Force all error messages to occur**
- **Apply inputs that force default values**
- **Explore character sets and data types**
- **Overflow input buffers**
- **Find inputs that may interact**
  - **test various combinations of their values**
- **Repeat the same inputs many times**

Output Constraint Attacks

AET, Inc
Advanced Engineering Technology

- **Force different outputs for each input**
- **Force invalid outputs**
- **Force output size change**
- **Force output to exceed output space**
- **Force the screen to refresh**

Storage Constraint Attacks

AET, Inc
Advanced Engineering Technology

- **Apply inputs under differing initial conditions**
- **Data Structure Over/Underflow**
  - **Force a data structure to store too many or too few values**
- **Find alternate ways to violate internal data constraints**

Computation Attacks

AET, Inc
Advanced Engineering Technology

- **Experiment with invalid operand and operator combinations**
- **Force a function to call itself recursively**
- **Force computation results to be too large or too small**
- **Find features that share data or interact poorly**

Conclusions

AET, Inc
Advanced Engineering Technology

- **Exploratory testing familiarizes testers with functionality**
- **Staging attacks means formulating clear goals**
- **Testing is fun**
- **Eighteen attacks establishes quality**

How to Break Software

James A. Whittaker and Alan A. Jorgensen
Florida Tech—Software Engineering
150 West University Boulevard
Melbourne, Florida 32901
jw@cs.fit.edu

Advanced Engineering Technology, Inc.
1900 S. Harbor City Blvd., Suite 115
Melbourne, Florida 32901aaj@aet-usa.com

Abstract

This paper presents a series of exploratory testing 'attacks'. These attacks are based upon a software failure model presented by the authors at a prior STAR EAST conference: Software developers routinely fail to constrain inputs, outputs, storage, and computations. This model has been shown to be effective for developing 'good' test cases. In this paper, we present testing attacks based upon this software failure model. We have included reproduction sequences for example bugs found in production software using these exploratory attacks.

Introduction

Understanding what software does—and how it might fail doing it—is a crucial part of being an effective tester. Our paper, "Why Software Fails", presented at STAR EAST 1999 presents a fault model that provides insight into developing such an understanding; however, a lot of practice is needed to put the fault model to good use.

In this paper, we present strategies for applying the fault model and provide detailed examples finding bugs in released production software. We call these various strategies for applying the fault model "attacks" because we think the mindset of attacking the software is an effective tool for learning testing techniques. In the academic environment in which we work, it is a good way to convince budding developers that testing is a fun way to spend one's time.

This paper covers attacks that are orchestrated from the user interface. The techniques discussed are also applicable to other interfaces of the software under test. These other interfaces include the file system, other applications, and the operating system itself. (Yes, all of these interfaces are actually operating system interfaces, but each differs in the potential methods for accessing the software under test via that interface).

We divide the attacks into four categories, one for each of the capabilities of software: accepting input, producing output, storing data and performing computation. Each of these corresponds to the four categories of failures in our failure model. Ideally, testers should apply the attacks one at a time. This

allows concentration on one single point of attack. Once the first attack has been executed, the tester can move to the second attack and so on. There are eighteen different attacks described below.

Exploratory Testing and Failure Models

One of the main problems in testing is that most applications are written without clear, documented requirements and testers almost always lack good specifications of behavior. This is very unfortunate, but the expectations are that we must test the product anyway. Obviously, this puts us in a tremendously difficult situation: we must test a product whose purpose is unclear (because we lack good requirements) and whose expected behavior is anyone's guess (because no written specification exists).

We encourage all testers to get as much information as possible about the application domain and the behavior of the software under test. User guides, competing products, help files and prior versions of the same product are all useful. We are not going to whine about the lack of good specs, however. Absent of specifications is a fact of life and other authors have written extensively on ways to get organizations to write and maintain requirements. Even when specifications exist, however, they are usually incomplete. What specifications have you read that detailed the behavior of a program in the circumstance of a coding error? Our purpose is to provide insight into effective testing even when there is insufficient documentation.

The most unfortunate result of a lack of documentation is to leave testers in the dark. Instead of being able to read about a program's behavior, they must spend time learning the application by using it. Too often, this leaves inexperienced testers in the "bang on the keyboard" mode, hoping to break the system. What is needed is an approach that includes this exploration process, but also leads to early and useful bug detection. We base our approach on what we call a "failure model." A failure model is a generalized description of the kinds of failures that we expect to find in the software. How do we determine the types of failures that we expect? We study the kinds of mistakes that programmers typically make. One such study resulted in our paper, "Why Software Fails" presented at STAR EAST, 1999 and published in ACM SE Notes **[1]**.

In "Why Software Fails" we determined that after software had been tested and released there still existed a major class of defects summarized by failure to properly constraint the software. In fact, there were four classes of constraint failures: 1) Input, 2) Output, 3) Storage, and 4) Computation. This should not be surprising since these are the four things that programs generally do. For details, please refer to our paper on the subject. For our purposes here, let us assume that all software may suffer from these four classes of failures and use this as a model to design attacks against the software. What this means is that we will assume that failures of these types exist and, while we are exploring the functionality of the software we are testing, we will also check for these types of failures. For each category of failure, there are several attacks, and, to date, we have found eighteen different attacks. We are very interested in hearing from you should you find yet another category of attack.

Input Constraint Attacks

Inputs need not be applied in an ad hoc manner. Instead, every input applied should either be part of an effective attack or it should be part of exploring the functionality in order to plan an attack. In the latter case, endeavor to *act like a user* to the best of your knowledge of how users will use the application. In other words, try to apply inputs that force the application to *get real work done*. When testing a word processor, create, format, and edit a document. Put yourself in the user's shoes and try to get done what the user will be doing with the application. Once the functionality is understand sufficiently, it's time to get nasty. Here's our advice on how to accomplish this.

First attack: Apply inputs that force all the error messages to occur

I like to start with erroneous conditions to get them out of the way. This is the low-hanging fruit of bugs. Pick it first and then move on to more sophisticated attacks. The idea is to make sure that the software doesn't try to process bad data. Enter values that are too big, too small, too long, too short, out of the

acceptable range, or of the wrong data type. When an error message occurs, determine the boundary conditions of the error. We have found an error in some applications that mark a file name too lengthy; however, at just the right length, the file name is accepted but the appended file name extent is truncated inappropriately.

Don't go overboard though, future attacks will round out the completeness of your tests, this first attack should ensure that you see each error message that the application can generate *at least once*. Once you've seen a message move on to another one.

The reason that this is an effective attack is that error cases require developers to write additional error-checking code. Writing such conditions often means that the developer has to stop thinking about the main-line functional code and consider erroneous data. This redirection of design activity is often done sloppily. Worse, many developers postpone writing the error code and then never get back to it.

It's amazingly difficult to make a program fail gracefully, and such difficulty usually means bugs. Some error messages are no-brainers; the program simply pauses execution to display the message and then continues on to the next input or timer expiration. However, other error messages result from an exception being thrown and an exception handler being executed. Exception handlers (or any centralized error routine) are problematic because the instruction pointer changes abruptly without corresponding changes to the data state. Suddenly, the exception handler is executing and all kinds of data problems can ensue: files could still be open, memory could still be allocated, data could remain uninitialized. When control once again returns to the main routine, it is hard to say at what point the error handler was called and what leftover side-effects might be waiting to trip up unwary developers: opening a file could fail because the file might already be open, or data might be used without proper initialization. If we ensure that we've seen all the error messages and the system still works well, we've done a huge service to our users (not to mention our maintenance developers).

The following procedure shows an interesting bug an FIT student found in Microsoft Word 2000 in which an error message appeared twice in a row for no particular reason. Certainly this is not a devastating error on the part of the developers, but it is very annoying to the user. This is one example of the type of bug you'll encounter as you apply this attack. Other more serious bugs are unhandled exceptions and error cases, as well as misleading error messages.

Try this:

(In this and each of the following examples, we use the notation "->" to mean "Click on." Each of these issues was reproduced using Windows NT with Service Pack 6a.)

1) Invoke MS Word 2000, Rev 9.0.2720
2) ->File->New->Blank Document
3) ->Insert->Index and Tables
4) Be sure that the "Index" tab is selected.
5) Double click the "Columns" entry field.
6) Enter the value "5"
7) -> OK
8) Note the error message indicating that the value must be between 0 and 4. (What is an index with 0 columns?)
9) -> OK
10) Note that the same error message reappears.

Two error messages appear to annoy and inconvenience the user.

Second attack: Apply inputs that force the software to establish default values

This is a wonderful attack because often it means doing nothing except clicking "OK" and watching the application die. Why would something so simple constitute an effective attack? Actually, just because the tester does nothing does not mean the software doesn't have to do work. In fact, establishing defaults is a

fairly intricate programming task. Developers have to make sure that variables are initialized before a loop is entered or before a function call is made. If this doesn't happen, then an internal variable might be used without being initialized. The result is often catastrophic.

For example, in Word 2000 the following dialog has an options menu that, when left unchanged, actually makes controls disappear when the dialog is redisplayed. Note the missing heading level controls. Moreover, it changes the display from 3 headings to 9 heading without the user entering any values!

Try this:

For this bug it is important that you start with a fresh copy of Word.

1) Invoke MS Word 2000, Rev 9.0.2720
2) ->File->New->Blank Document
3) ->Insert->Index and Tables
4) -> "Table of Contents" tab.
5) -> Options
6) -> OK (Changing nothing)
7) Note that everything is normal. There are 3 levels selected and three levels displayed.
8) Repeat steps 5 and 6.
9) Now note that there are 9 levels selected!
10) Repeat steps 5 and 6.
11) Now note that the levels control has disappeared!

The default settings change without warning, forcing the user to reset the values.

This odd behavior nicely demonstrates the second attack: force software to display the value of internal data and then change some (but not all) or none of the values. This requires defaults to be set; if no defaults have been coded, then the software may very well fail.

Sometimes forcing defaults requires changing values from their initial settings once and then changing them a second time to an improper configuration. These back-to-back changes ensure that the default settings can be re-established once they are changed to other valid values.

Third attack: Explore allowable character sets and data types

Some input values are simply problematic, particularly when you consider that special characters like $, %, #, quotation marks and so forth have special meaning in many programming languages and often require special handling when they are read as input. If the developer failed to consider this situation, then these inputs may cause the program to fail when they are encountered. Sometimes historic reserved words may cause a problem.

Fourth attack: Overflow input buffers

The idea here is to enter long strings in order to overflow input buffers. This is a favorite attack by hackers because, sometimes, the application is still executing a process after it crashes. If a hacker attaches an executable string to the end of the long input string, the process may execute it.

A buffer overflow in Word 2000 is one such exploitable bug. The bug is in the Find/Replace feature and is shown below. It is interesting to note that the "Find" field is properly constrained but the "Replace" field is not.

Try this:

***WARNING:** Don't try this when you are doing anything else useful on your machine!*

1) Invoke MS Word 2000, Rev 9.0.2720
2) ->File->New->Blank Document

3) ->Edit->Find->Replace
4) Enter “abcd” [Tab] selecting the “Replace” field.
5) Enter “1234567890”
6) Type [Shift+Ctrl+Home] [Ctrl+C] placing “1234567890” in the clipboard.
7) Type [Ctrl+V] 30 times attempting to place 300 characters in the replace buffer.
8) ->“Find Next”
9) Note that one of two things happens: If you are lucky you will get an MS Word Application Error. If you are unlucky, the MS Word application will die but the winword process will still be running and you will need to reboot.

Why the “Find” field is length-constrained and the “Replace” field is not is a real mystery.

Security concerns top the list of reasons why testers should take buffer overruns seriously. However, we notice that all too often novice testers go completely overboard with this attack. We often have to tell our students to stop long string attacks and try something else. The reason is that many developers feel they are low-probability events and will not fix them. Consequently, it is good to get an idea about what developers feel are reasonable lengths for input fields before wasting a lot of time on this attack.

Fifth attack: Find inputs that may interact and test various combinations of their values

Up to now, we have only dealt with attacks that exploit a single input entry point into the software. In other words, we have been picking an input location and poking it until the software breaks. This next attack deals with multiple inputs that are processed together or that influence one another. For example, an API that can be called with two parameters requires selection of values for one parameter based on the value chosen for the other parameter. It is often the combination of values that was programmed incorrectly because of the complexity of the logic involved in coding the solution.

As an example, try the following with Word 2000. Choose the Insert option from the Table menu and experiment with the allowable values for the number of columns and the number of rows. You will soon realize that these input fields cannot be overflowed and that the maximum number of columns is 63 and the maximum number of rows is 32767. This is a good example of input value dependence. If you enter small numbers for both then Word handles this just fine. A large number for one and a small number for the other is also fine. But if you enter the maximum values of 63 and anything above 32000 for the number of rows, the application hangs because it overwhelms the machine’s CPU cycles.

Try this:

1) Invoke MS Word 2000, Rev 9.0.2720
2) ->File->New->Blank Document
3) ->Table -> Insert -> Table
4) Enter 63 in the Number of Columns field
5) Enter 32000 in the Number of Rows field (Doesn’t this seem like a lot of rows to allow?)
6) -> OK
7) Be prepared to wait some period of time. Hmmm. Nothing happening? Better check with Task Manager.

Breaking some dialogs requires that we concentrate on the relationship between multiple inputs.

Which combinations are problematic? This is an issue still being actively researched, but an approach we have found to be especially effective is to select an output and then find input combinations that cause that output to occur.

Sixth attack: Repeat the same input or series of inputs numerous times

Repetition often has the effect of gobbling resources and stressing an application’s stored data space, not to mention uncovering undesirable side-effects. Unfortunately, most applications are unaware of their own space and time limitations and many developers like to assume that plenty of resources are always

available.

An example of this can be found in Word's equation editor that seems to be unaware that it can only handle 10 levels of nested brackets. Indeed, after the 10^{th} pair of brackets, the equation disappears.

Try this:

1) Invoke MS Word 2000, Rev 9.0.2720
2) ->File->New->Blank Document
3) -> Insert -> Object -> Microsoft Equation 3.0
4) In the Equation dialog box, Click on ( ) and ( ) to insert a parenthesis pair.
5) Repeat step 4) ten more times.
6) Note the error message, "Can't have more than 10 levels of nested templates, matrices, and/or embellishments.
7) -> OK
8) -> Edit -> Select All
9) Repeat step 4.
10) Note the Error message of Step 6).
11) Also note that the original equation entered at step 5) is now missing.

Repeating the same input or sequence of inputs over and over will often cause strange behavior.

Output Constraint Attacks

Applying inputs is fairly straightforward and, unfortunately, many testers equate testing with applying a large number of different inputs and input combinations. Our research indicates, however, that many bugs are simply too difficult to find by concentrating on inputs alone. Instead, we take the more difficult approach of beginning with software *outputs* and work our way back to causal inputs.

The next series of attacks will require us to identify interesting outputs and figure out which inputs are capable of driving the application to generate those outputs. Attacks 7-11 give us insight into how we go about selecting which outputs to concentrate on.

Seventh attack: Force different outputs to be generated for each input

It is often the case that a single input can generate any number of outputs, depending on the context under which it is applied. For example, if we must test a telephone switch, one input that must be tested is the ability of the switch to correctly process the input "the user picks up the phone." Since there are two major outputs that the switch will generate when this input is applied, we must test them both. Consider first the case that the phone is idle and the user picks up the receiver: the switch will generate a dial-tone output and send it to the user's phone. Now consider the case in which the phone is ringing: the switch will connect our user with the other subscriber who placed the call. Thus, we have tested the two major outputs (or behaviors) associated with the user picking up the telephone receiver.

Identifying all possible outputs for the most important or frequently used inputs is an important exercise. Ensuring that testing covers each of these outputs can be hard work, but will pay off by helping us find important bugs that will irritate our users.

Eighth attack: Force invalid outputs to be generated

This is a very effective attack for testers who really understand their problem domain. For example, if you are testing a calculator and understand that some functions have a restricted range for their result, trying to find input value combinations that force that result is a worthwhile effort. However, if you do not understand mathematics, it is likely that such an endeavor will be a waste of time—you might even

interpret an incorrect result as correct.

One of our favorite bugs falls into this category. A Y2K-related bug in Windows NT (which was fixed in service pack 5) actually allowed the system to display the date February 29, 2001—an invalid output because the year 2001 is not a leap year. In this case, a tester unfamiliar with the leap year rule would undoubtedly have missed this bug.

Ninth attack: Force output size or dimension to change

Another useful attack along these same lines is forcing a complex output to be generated and then changing some property of the output. The property that is often the most convenient for user interface testing is output size, i.e., force display areas to be recomputed by changing the length of inputs and input strings.

A good conceptual example is setting a clock to 9:59 and watching it roll over to 10:00. In the first case the display area is 4 characters long and in the second it is 5. Going the other way, we establish 12:59 (5 characters) and then watch the text shrink to 1:00 (4 characters). Too often developers write code to work with the initial case of a blank display area and are often disappointed when the display area already has data in it and new data of different size is used to replace it.

As an example, "WordArt" in PowerPoint has an interesting problem. Suppose we enter a long string as illustrated by the test sequence below.

Notice that two things occur when the OK button is pressed. First, the routine computes the size of the output field need and then populates the field with the text we entered.

Next we will edit the string and replace it with a single character. Notice that the display area stays the same size despite the fact that only one character is inserted and the font size is not changed. Further, if we edit the string again and type a multi-line string the output is even more interesting.

Try this:

1) Invoke Microsoft Power Point Version 9.0.2716.
2) ->OK->OK selecting a new blank presentation.
3) ->Insert->Picture->Word Art
4) Select any Word Art Shape -> OK
5) Type "This is very long text meant of overflow the screen so that not all of it can be seen."
6) ->OK
7) In the Word Art window -> Edit Text
8) Enter the letter "a"
9) -> OK
10) Note the long stretchy 'a"
11) Repeat step 7
12) Enter "This is very long text meant of overflow the screen." Three times.
13) Note the unusual but pretty effect.

The application forgot to resize the text box when the text was reformatted.

I think the point is made and we can move on to the next attack.

Tenth attack: Force output to exceed the size of its destination

This is another attack based on outputs that is very similar to the previous attack. However, instead of looking for ways to cause the area inside the display to get corrupted, we are going to concentrate on the area outside the display. This time we are going to do things we hope don't require recalculation of the display boundaries but simply overflow them.

Considering PowerPoint again, we can draw a textbox and fill it with a superscripted string. Then, changing the size of the superscript to a large font causes the top of the exponent to be truncated.

Try this:

1) Invoke Microsoft Power Point Version 9.0.2716.
2) ->OK->OK selecting a new blank presentation.
3) Click on the first text box.
4) Enter “This is something else “
5) ->Edit->Select All
6) ->Format->Font
7) In the Font dialog box select “Superscript”
8) In the font size box enter “80”
9) -> OK
10) Note that that tops of the characters are missing.

The text gets chopped at the top because it is too big for the text box.

Eleventh attack: Force the screen to refresh

Refreshing the screen or window as a result of a user applying some input is a major problem for users of modern windows-based GUIs. It is an even bigger problem for developers: refresh too often and you slow down your application, failing to refresh causes anything from minor annoyances (i.e., requiring the user to force refresh) to major bugs (preventing the user from getting work done).

The general idea in searching for refresh problems is to add, delete and move objects around on the screen. This causes the background object to redisplay, and if it doesn’t do it properly and in a timely fashion, you have just found the classic refresh bug. It is a good idea to try varying the distance you move an object from its original location. Move it a little, then move it a lot; move it once or twice, then move it a dozen times.

Continuing with the large superscript example from above, try moving it around on the screen a little at a time. Note the nasty refresh problem shown below.

Try this:

1) Invoke Microsoft Power Point Version 9.0.2716.
2) ->OK->OK selecting a new blank presentation.
3) -> Insert -> Text Box
4) Click somewhere on the slide.
5) Enter “a9” <Shift+Backspace> to highlight the ‘9’ (Omit the quote marks).
6) -> Format -> Font
7) Select the “Superscript” option
8) Select the “Size” field.
9) Enter “80”
10) -> OK
11) Note that the top part of the ‘9’ does not display.
12) Move the cursor to the edge of the text box so that the quad arrow cursor appears.
13) Click and Drag the text box downward about half the length of the text box.
14) Note artifact left on the screen, the missing top part of the ‘9’.

The application has no idea that it is leaving behind a trail of dangling text fragments.

Another recurring problem in Office 2000 associated with screen refresh is disappearing text. This is most annoying in Word just around the page and paragraph formatting boundaries.

Storage Constraint Attacks

Data is the lifeblood of software; if you manage to corrupt it, the software will eventually have to use the bad data and what happens then may not be pretty. It is worthwhile to understand how and where data values are established.

Essentially, data is stored either by reading input and then storing it internally or by storing the result of some internal computation. By supplying input and forcing computation, we enable data flow through the application under test. The attacks on data follow this simple fact as outlined in attacks 12-14. However, without access to the source code, many testers do not bother to consider these attacks. We believe, though, that useful testing can be done even though specifics of the data implementation are hidden. We like to tell our students to practice "looking through the interface." In other words, take note of what data is being stored while the software system is in use. If data is entered on one screen and visible on another, then it is being stored. Information that is available at any time is being stored.

Some data is easy to see. A table structure in a word processor is one such example in which not only the data but the general storage mechanism is displayed on the screen. Some data is hidden behind the interface and requires analysis to discover its properties.

Once the nature of the data being stored is understood, try to put yourself in the position of the programmer and think of the possible data structures that might be used to store such data. The more that programming and data structures are understood, the easier it will be to execute the following attacks. The more completely you understand the data you are testing, the more successful the attacks will be at finding bugs.

Twelfth attack: Apply inputs using a variety of initial conditions

Inputs are often applicable in a variety of circumstances. Saving a file, for example, can be performed when changes have been made, and it can also be performed when no changes have been made. Testers are wise to apply each input in a number of different circumstances to account for the many such interactions that users will encounter when using the application.

Thirteenth attack: Force a data structure to store too many/too few values

There is an upper limit on the size of all data structures. Some data structures can grow to fill the capacity of machine memory or hard disk space and others have a fixed upper limit. For example, a running monthly sales average might be stored in an array bounded at 12 or fewer entries, one for each month of the year.

If you can detect the limits on a data structure, try to force too many values into the structure. If the number is particularly large, the developer may have been sloppy and not programmed an error case for overflow.

Special attention should be paid to structures whose limits fall on the boundary of data types 255, 1023, 32767 and so on. Such limits are often imposed simply by declaration of the structure's size and very often lack an overflow error case.

Underflow is also a possibility and should be tested as well. This is an easy case, requiring only that we delete one more element than we add. Try deleting when the structure is empty, then try adding an element and deleting two elements and so on. Give up if the application handles 3 or 4 such attempts.

Fourteenth attack: Investigate alternate ways to modify internal data constraints

The phrase "the right hand knoweth not what the left hand doeth" describes this class of bugs. The idea is simple and developers leave themselves wide open to this attack; in most programs there are lots of ways to do almost anything. What this means to testers is that the same function can be invoked from numerous

entry points, each of which must ensure that the initial conditions of the function are met.

An excellent example of this is the crashing bug one of our students found in PowerPoint, regarding the size of a tabular data structure. The act of creating the table is constrained to 25×25 as the maximum size. However, one can create such a table, then add rows and columns to it from another location in the program—crashing the application. The right hand knew better than to allow a 26×26 table but the left hand wasn't aware of the rule.

Try this:

1) Invoke Microsoft Power Point Version 9.0.2716.
2) ->OK
3) Select the black slide.
4) ->OK selecting a new blank presentation.
5) -> Insert -> Table
6) Enter "26" <tab> "26"
7) -> OK -> OK
8) Note that the rows and columns default to 25 x 25.
9) -> Table -> "Insert Columns to the Left" on the Tables and Borders dialog box.
10) -> Table -> "Insert Rows Above" on the Tables and Borders dialog box.
11) Now wait. Tap…Tap…Tap… Hmmm. Tap…Tap…Tap….
12) Try <Ctrl+Shift+Esc>

Computation Constraint Attacks

Computation, using both data that is stored internally and data that is received as input from users, is one of the most fundamental tasks that software performs, and it presents challenging testing problems. Like data, computation cannot be directly seen; it is hidden behind the user interface and much of the details associated with a particular computation must be surmised without benefit of the source.

Computation is also everywhere in a software application. Computation is performed in assignment statements that pervade all code, no matter its functionality. Software computes when it loops, computes when it branches and computes when its features interact with its stored data. The next three attacks will, hopefully, put some perspective on this difficult testing endeavor.

Fifteenth attack: Experiment with invalid operand and operator combinations

This class of attacks requires investigation of the data type and allowable values associated with operands in one or more internal computations. If one has access to the source, this information is obtainable. Otherwise, testers must do their best at determining what computation is taking place and what type of data is being used.

Sometimes inputs or stored data are well within the legal boundaries but are illegal for some types of computation. Division by zero is a good example. Zero is a valid integer, but it is invalid as the denominator of a division computation.

Computations that have more than one operand are subject to not only the above attack but also to potential operand conflict. For example, both character and number types can be combined with the '+' operator in many programming languages. In the former case, adding characters causes them to be concatenated, and in the latter case, integer arithmetic is performed. However, forcing a software system to add a character to a number (conflicting operands) might cause a failure.

Sixteenth attack: Force a function to call itself recursively

Functions often call other functions to get work done. Sometimes, functions call themselves. This is called *recursion* and it is a powerful alternative for iterative loops that developers often employ. Both loops and recursion can be problematic if the number of times they execute is not limited to a finite number. The "infinite loop" is a common programming error; however, such things are generally well covered in unit testing. As a system tester, it has been many years since I saw such a problem remain unfixed long enough to be seen.

Recursion, however, is another story altogether. Modern software applications offer ways for objects to reference themselves, which, in turn, offers testers new ways to break them.

The hyperlink is the most common analogy. Imagine a web page that has a link to itself. This is the general idea of recursion. Now imagine a web page with a script that automatically executes when the page is displayed. Suppose that script reloads its host page, which will execute the script, which reloads the host page, which executes the script… You get the idea. This shows the danger of recursion. If it is implemented improperly, it will quickly overwhelm the resources of the machine and eventually generate a heap overflow.

Seventeenth attack: Force computation results to be too large or too small

The next class of computation attacks is aimed at overflowing and underflowing data objects that store computation results.

Even simple computations like `y=x+1` are problematic around boundary values. If both `x` and `y` are signed 2 byte integers and `x` has the value 32767, this computation will fail because the result will overflow its storage; the result exceeds the range of acceptable signed 2 byte integers.

The same thing goes at the negative end of a data type. `y=x-1` will fail if we can assign `x` the value -32768.

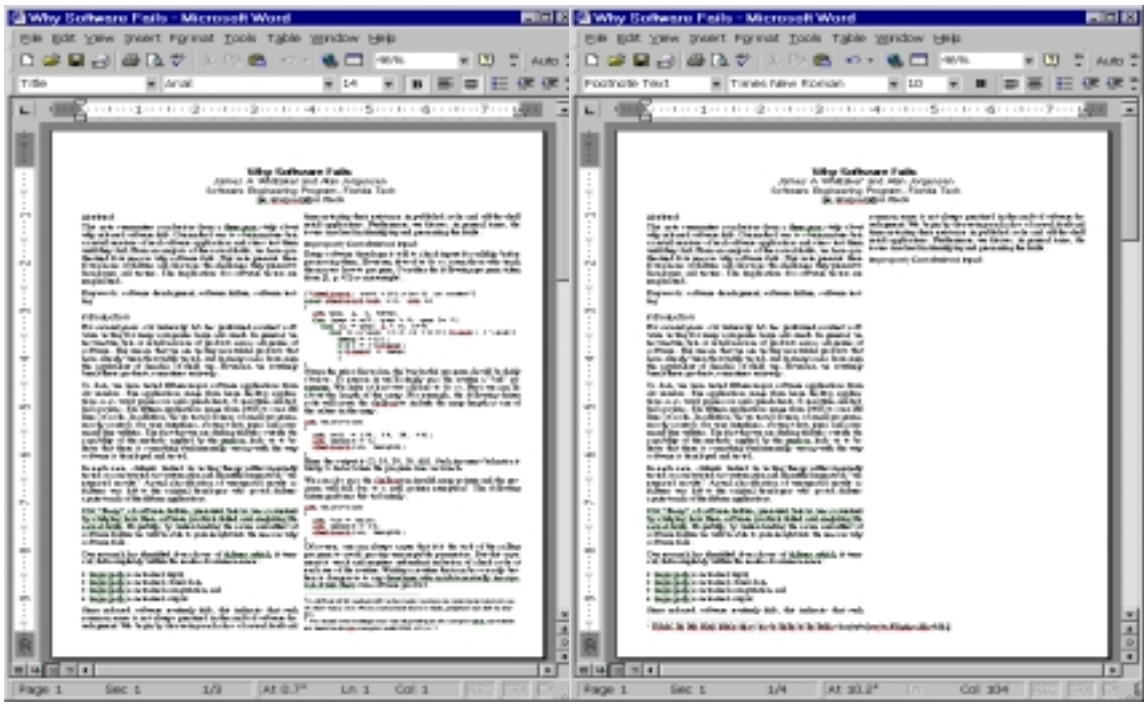
Eighteenth attack: Find features that share data or interact poorly

The last attack category discussed in this paper is perhaps the granddaddy of them all and the one that separates testing novices from the pros: feature interaction. The problem here is nothing new: different application features share the same data space and the interaction of the two features causes the application to fail.

Features that share data and could interpret that data in conflicting ways provoke an open question: How do you test feature interaction? Right now we are stuck with trial and error. So this example must suffice for now.

This example shows an unexpected result when combining footnotes and dual columns on a single page in Word 2000. The problem is that Word computes the page width of a footnote from the reference point of the note. Thus, if one has two footnotes on the same page, one referenced from a dual column location and one from a single column location, the single column footnote pushes the dual column footnote to the next page. In addition, any text between the note's reference point and the bottom of the page is pushed to the next page.

The following screen shots illustrate the problem vividly. Where is the second column of text? (It is on the next page along with the footnote.) Can you live with the document looking like this? You must, unless you find a workaround (which means time spent away from preparing your document).



Other examples in Word 2000 include problems with widow/orphan control on paragraphs with embedded pictures and resizing text boxes that have been grouped with other types of objects.

Conclusions

Simply going through the eighteen attacks outlined above can exercise a great deal of an application's functionality. Indeed, staging a successful attack usually means experimenting with dozens of possibilities and pursuing a number of dead-ends. Just because some of this exploration doesn't find bugs, however, does not mean that the technique is not useful. First of all, the time spent using the application familiarizes testers with the range of possible functionality and leads to new ideas for additional attacks. Secondly, unsuccessful tests (tests that find no bugs) are good news! They are an indication of product reliability. This is particularly so if the set of tests is the set of malicious attacks outlined in this paper. If code can withstand this treatment, it may very well withstand whatever users can dish out.

Never underestimate the value of having a concrete goal in mind when testing. We've seen too many testers waste time poking at a keyboard or making random API calls hoping something breaks. Staging attacks means formulating clear goals—based specifically on things that *could* go wrong—and then designing the tests that investigate those goals. This way, every test has a purpose and progress can be readily monitored.

Finally, remember always that testing should be fun. It certainly is for us. The attack analogy supports this good-natured view of testing and adds a little more spice to a very enjoyable pastime. Happy hunting!

References

1. Whittaker, James A. and Alan A. Jorgensen, “Why Software Fails.” ACM Software Engineering Notes, July 1999. Also awarded “Best Presentation” at STAR EAST 1999, Orlando, Florida.

2. Marick, Brian, “A Survey of Exploratory Testing**”,** http://www.testingcraft.com/exploratory.html,

Wednesday 6 December 2000

Keynote 2

How to Break Software (with examples)

Alan Jorgensen

Alan Jorgensen began programming in 1959 and wrote his first test automation software in 1963. A considerable portion of his career has been spent testing (and troubleshooting) real-time process control systems such as those in nuclear power stations. He is currently developing a model based software test automation system, Software Pest Control Suite (SPCS), and is an Adjunct Professor at Florida Institute of Technology where he teaches Computer Science and Software Engineering. Once in a while he goes fishing along side the alligators.