Presentation Notes
Paper
Bio
Return to Main Menu

PRESENTATION

WG2

Wednesday, November 3, 1999
10:30 AM

# WHY SOFTWARE FAILS (AND HOW TESTERS CAN EXPLOIT IT)

**James Whittaker**
Florida Institute of Technology

INTERNATIONAL CONFERENCE ON
**SOFTWARE TESTING, ANALYSIS & REVIEW**
NOVEMBER 1-5, 1999
SAN JOSE, CA

# Why Software Fails
# (and how testers can exploit it)

Dr. James A. Whittaker

# Formal Methods

“If only you did things more formally…

Formal methods are an error-prone way
to prove toy programs correct.
They are simply no match for good testers.

…your code would be better.”

# Tools

“If only you used the right tools…

CASE tools are probably the lowest quality software ever written. Using them means inheriting their bugs. Testers rejoice!

…your code would be better.”

# Process Improvement

“If only you worked more carefully…

SPI is a management technique,
software development is a technical problem.
You cannot manage-away technical problems.
A CMM rating won’t protect you from testers.

…your code would be better.”

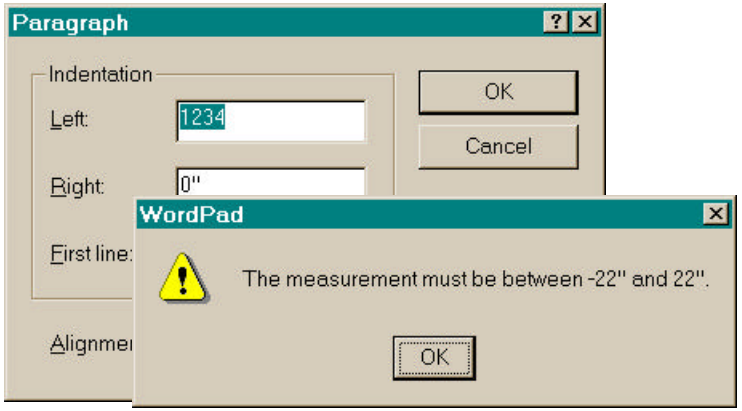
# The Threat of Destruction

"If only you paid attention to your testers…

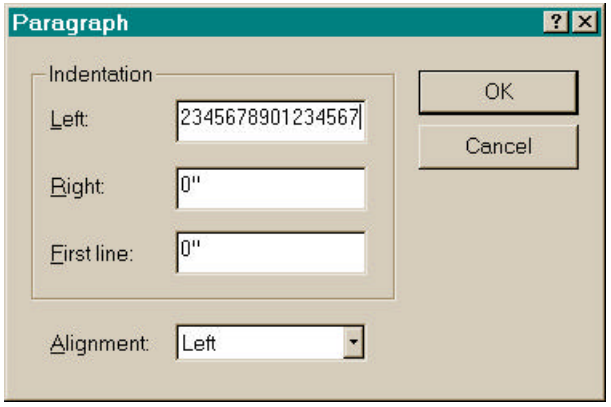
We're the ones who break things.
We're the ones who can show you the problems.
Listen to us or face our wrath!

…your code would be better."

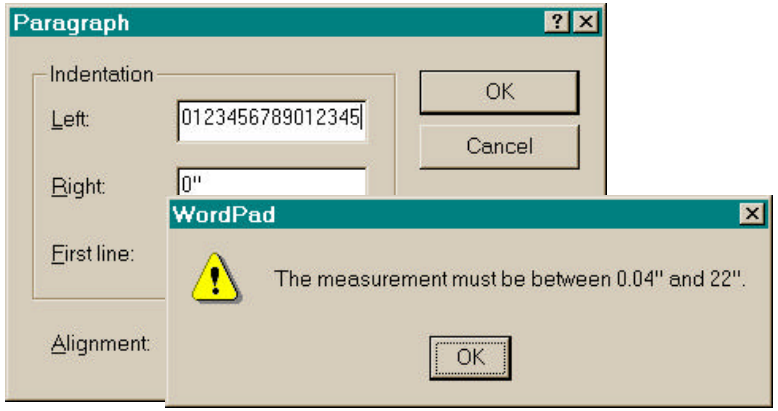
# Exploiting Broken Input Constraints
## (WordPad for Windows NT 4)



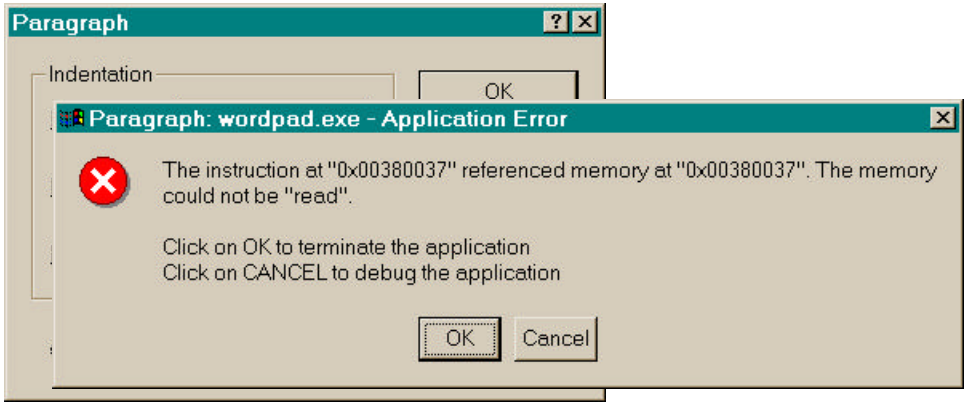

(a) The program correctly constrains an input with 4 digits




(b) The program fails when it accepts a 17 digit number which is out of range



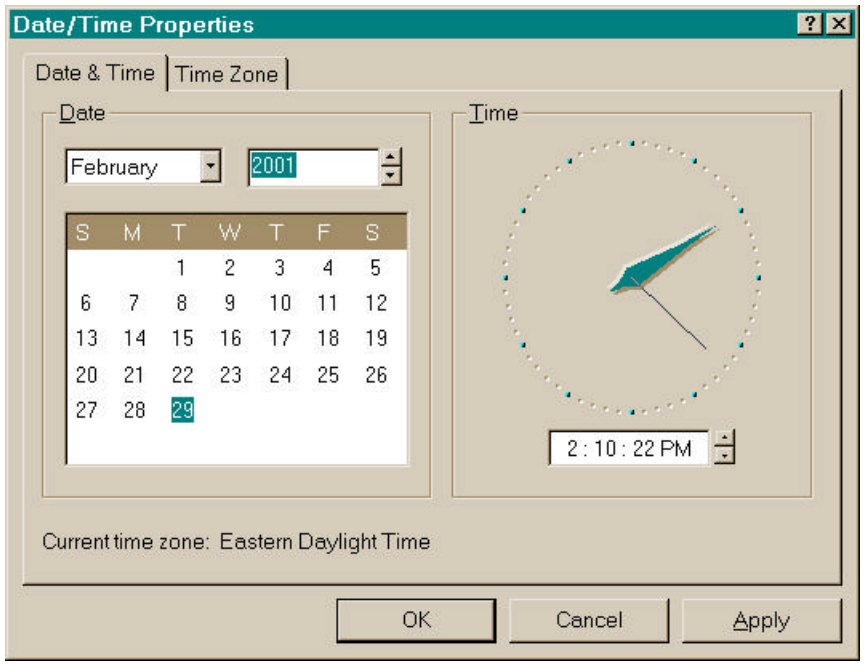

(c) The rules about allowable inputs change when a 65 digit number is entered




(d) Finally the application crashes on a 70 digit number

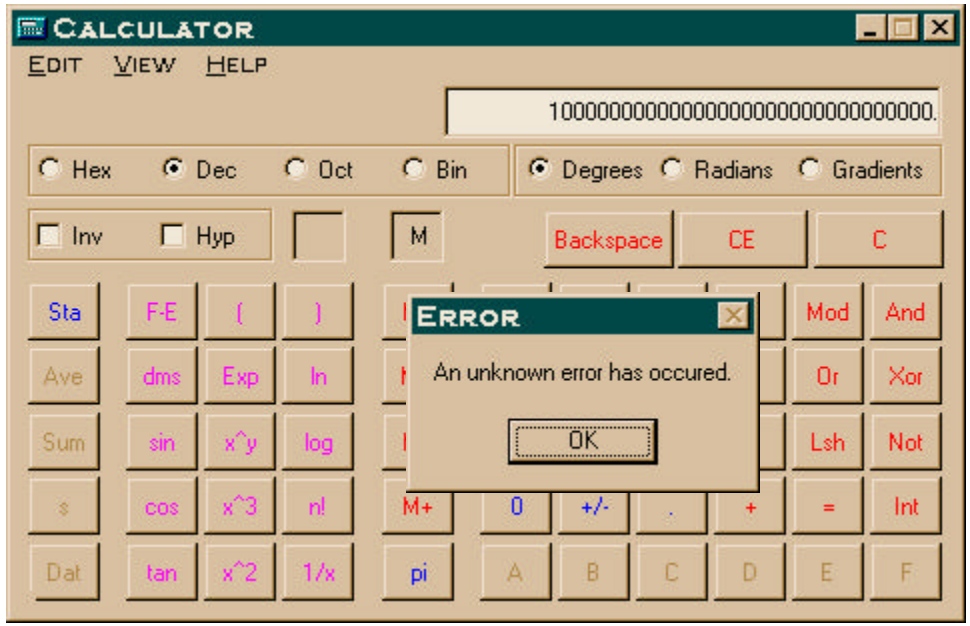
# Exploiting Broken Output Constraints

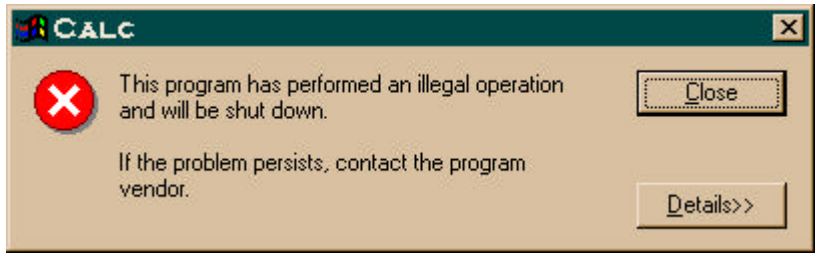
(Time/Date Properties for Windows NT 4)



# Exploiting Broken Data Constraints

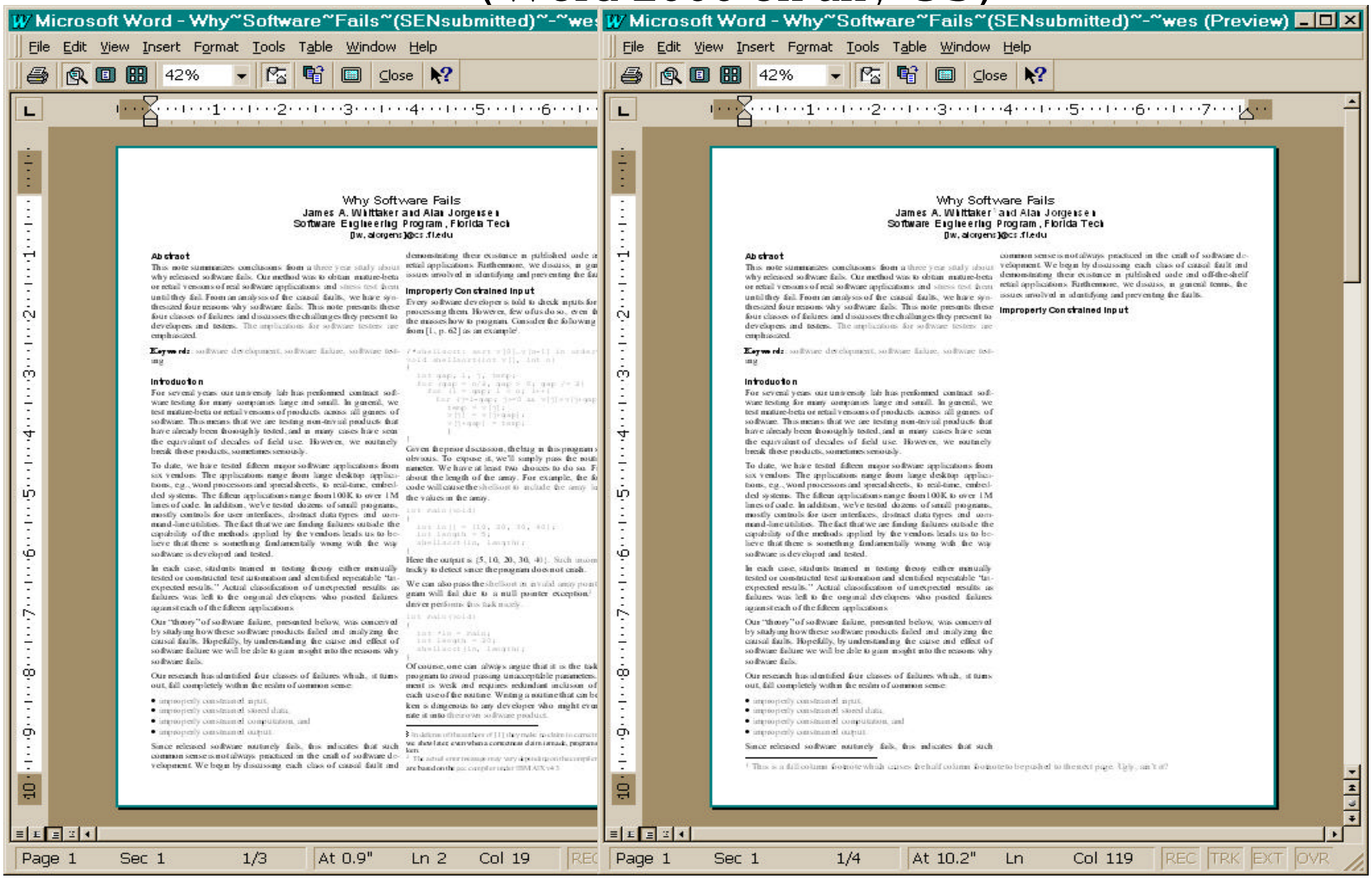
(Calculator for Windows 98)



# Exploiting Broken Computation Constraints

(Word 2000 on any OS)



# Conclusion

- I believe testers understand *why software fails* better than anyone
- My lab is dedicated to breaking software until we learn enough to fundamentally improve development practices
- Until such time, however, we are creating new testing technology and tools to improve finding and measuring faults and failures

# Why Software Fails

James A. Whittaker and Alan Jorgensen
Software Engineering Program, Florida Tech
[jw, ajorgens]@cs.fit.edu

## Abstract

This note summarizes conclusions from a three year study about why released software fails. Our method was to obtain mature-beta or retail versions of real software applications and stress test them until they fail. From an analysis of the causal faults, we have synthesized four reasons why software fails. This note presents these four classes of failures and discusses the challenges they present to developers and testers. The implications for software testers are emphasized.

**Keywords**: software development, software failure, software testing

## Introduction

For several years our university lab has performed contract software testing for many companies large and small. In general, we test mature-beta or retail versions of products across all genres of software. This means that we are testing non-trivial products that have already been thoroughly tested, and in many cases have seen the equivalent of decades of field use. However, we routinely break these products, sometimes seriously.

To date, we have tested fifteen major software applications from six vendors. The applications range from large desktop applications, e.g., word processors and spreadsheets, to real-time, embedded systems. The fifteen applications range from 100K to over 1M lines of code. In addition, we've tested dozens of small programs, mostly controls for user interfaces, abstract data types and command-line utilities. The fact that we are finding failures outside the capability of the methods applied by the vendors leads us to believe that there is something fundamentally wrong with the way software is developed and tested.

In each case, students trained in testing theory either manually tested or constructed test automation and identified repeatable "unexpected results." Actual classification of unexpected results as failures was left to the original developers who posted failures against each of the fifteen applications.

Our "theory" of software failure, presented below, was conceived by studying how these software products failed and analyzing the causal faults. Hopefully, by understanding the cause and effect of software failure we will be able to gain insight into the reasons why software fails.

Our research has identified four classes of failures which, it turns out, fall completely within the realm of common sense:

- improperly constrained input,
- improperly constrained stored data,
- improperly constrained computation, and
- improperly constrained output.

Since released software routinely fails, this indicates that such common sense is not always practiced in the craft of software development. We begin by discussing each class of causal fault and demonstrating their existence in published code and off-the-shelf retail applications. Furthermore, we discuss, in general terms, the issues involved in identifying and preventing the faults.

## Improperly Constrained Input

Every software developer is told to check inputs for validity before processing them. However, few of us do so, even those who teach the masses how to program. Consider the following program taken from [1, p. 62] as an example[1].

```
/*shellsort: sort v[0]…v[n-1] in order*/
void shellsort(int v[], int n)
{
  int gap, i, j, temp;
  for (gap = n/2, gap > 0; gap /= 2)
    for (i = gap; i < n; i++)
      for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap){
        temp = v[j];
        v[j] = v[j+gap];
        v[j+gap] = temp;
        }
}
```

Given the prior discussion, the bug in this program should be fairly obvious. To expose it, we'll simply pass the routine a "bad" parameter. We have at least two choices to do so. First, we can lie about the length of the array. For example, the following driver code will cause the shellsort to include the array length as one of the values in the array.

```
int main(void)
{
  int in[] = {10, 20, 30, 40};
  int length = 5;
  shellsort(in, length);
}
```

Here the output is {5, 10, 20, 30, 40}. Such incorrect behavior is tricky to detect since the program does not crash.

We can also pass the shellsort an invalid array pointer and the program will fail due to a null pointer exception.[2] The following driver performs this task nicely.

```
int main(void)
{
  int *in = main;
  int length = 20;
  shellsort(in, length);
}
```

Of course, one can always argue that it is the task of the calling program to avoid passing unacceptable parameters. But this argument is weak and requires redundant inclusion of check code at each use of the routine. Writing a routine that can be so easily

[1] In defense of the authors of [1], they make no claim to correctness. However, as we show later, even when a correctness claim is made, programs can still be broken.

[2] The actual error message may vary depending on the compiler used, our results are based on the gcc compiler under IBM AIX v4.3.

broken is dangerous to any developer who might eventually incorporate it into their own software product.

How does one systematically cause such failures? In our lab we have taken two approaches. From a black box perspective, we identify the places that input is allowed in the system. At each of these locations, we submit a large number of inputs of varying types, lengths and values. Sometimes conventional testing theory helps, for example, by aiding in the selection of boundary values for certain data types. However, our experience has been that not all failures occur at boundary locations. Thus, we are working on generalizing the selection of input values based on the failures we have elicited from the products we've tested. From a white box perspective, the task is easier because the source gives us easy access to inputs and data types. Automation of the identification of input entry points and of value selection is an area we are studying in earnest.

Consideration of the operating environment is also a crucial aspect of identifying inputs that cause the software to fail. Sometimes, an input that works in one environment might not work in another. Other times, resource contentions between various users (e.g., editing the same file) might elicit failures. Whenever we find a failure, we try to duplicate the erroneous behavior on different machines, different operating systems and vary what we call "background noise"—other applications that are running during a test that compete for resources.

Prevention of broken input constraints doesn't necessarily mean riddling source code with "if" statements to check value ranges. Often, it is simply a matter of tweaking the user interface to properly filter user input; other times calling a single routine that validates inputs can help source code remain clean.

There is nothing new about input constraint errors. Most good programming texts tell us to avoid them. The problem is that they fail to tell us how to do so and then proceed to show us code in which they, themselves, do not properly constrain inputs. If correct software really is important, we must begin to take our own advice seriously.

## Improperly Constrained Stored Data

Keeping bad input out of a software product is only one aspect of preventing bad data. Sometimes bad data is stored as a result of internal processing, causing the software to corrupt its own stored data. As a case-in-point, a commercial spreadsheet package can be tricked into this situation after entering a formula that is longer than the allowed limit.

When Microsoft® Excel® 97 receives a formula over 256 characters in length, it displays a message indicating that the formula is too long (see figure 1).

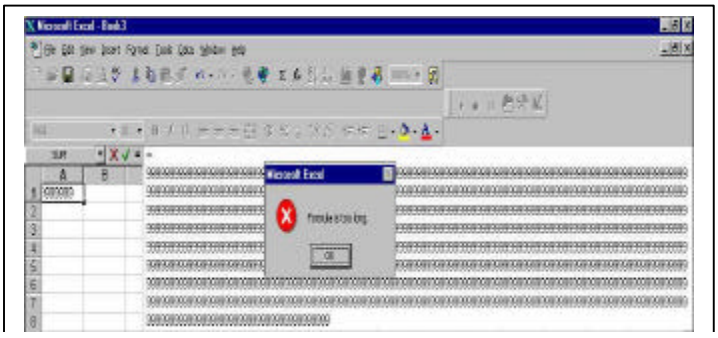


**Figure 1. Microsoft® Excel® 97 Correctly Constrains Input**

Thus, it successfully constrained the input and avoided the problem noted above. However, in doing so, it managed to corrupt its internal memory. Hitting the Enter key in the cell in which the formula was attempted causes the spreadsheet to completely crash (see figure 2).

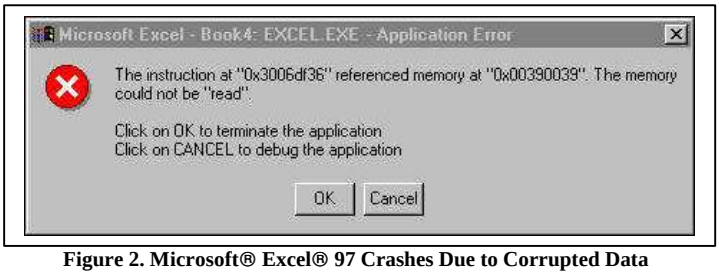



**Figure 2. Microsoft® Excel® 97 Crashes Due to Corrupted Data**

Input sequencing is an important aspect of breaking software by corrupting its internally stored data. It took two inputs in sequence to cause the spreadsheet to crash: the formula had to be followed by the Enter key. Some failures occur only after long, complex input sequences. This makes them difficult to reproduce and diagnose.

Very little testing technology exists for checking constraints on internally stored data and finding input sequences that violate them. To do so, models of sequences must be constructed based on the relationship of inputs and the knowledge of how data is stored in the system.

To model the relationship between stored data and input sequences, testers often create state-transition diagrams that describe how data flows through the system. The problem is that few systematic approaches to building enumerating software states and controlling the size of the state diagram exist.

Our experience has been that this class of problem creates very dangerous situations for users. Either the system completely crashes or it corrupts its internal data to the point that its results are no longer trustworthy. The fact that testing theory covers this subject so thinly makes this area ripe for additional research.

## Improperly Constrained Computation

The design for a running sales average is presented in [2, p. 11] and "verified" later in the same text [2, p. 117], including verification for "improper use." However, we can make this program fail by forcing it to overflow it's stored data through a simple calculation (that is "verified" correct).

Their running average is computed by :

$$R(i) = \frac{S(i) + S(i-1) + \ldots + S(i-11)}{12} \qquad (1)$$

where $S$ is an input (called a stimulus), $R$ the output (called a response) and $i$ the index representing the order of arrival of the inputs. When this program is implemented,[3] we can overflow the

[3] Many authors might argue that the presentation of an *algorithm* is exempt from the limitations of finite computers. However, we accept this argument only from authors who write about algorithms. If algorithms appear in a book about programming, then it isn't appropriate to ignore the physical machine and its limitations.

storage set aside for the running sum by submitting two or more inputs that, when added together, are larger than the maximum allowed integer.

Here we are exploiting the fact that this particular computation is unconstrained. In fact, there is no check to ensure that the result will fall within an acceptable range. To fix this problem we must check the values by subtracting the sum from the maximum allowable integer to ensure that the result is greater than or equal to the next input.

One could argue that such circumstances are unlikely, however, the result of overflow is almost always a system crash. For real-time software, this is a dangerous situation no matter how rare. Consider the aborted maiden flight of the *Ariane 5* rocket (see http://www.cs.wits.ac.za/~bob/ariane5.htm). The cause of this failure is similar to the running average problem above. A computation, in this case the conversion of a floating point number to an integer, produced a result that fell outside the allowable range. The rocket was destroyed in flight. Improperly constrained computation had a serious consequence.

The key to finding improperly constrained computation is knowledge of the problem domain. In an informal experiment at Florida Tech, we had a group of students test a desktop calculator. In every instance, students with substantial training in mathematics were able to discover many more problems in significantly less time than students with less mathematics expertise. By understanding the problem, one naturally has a better feel for the types of computation the software performs and is in a better position to diagnose possible problems.

In the absence of good domain expertise, it also helps to have the source code available. A simple search through the source looking for places where operators or built-in system functions are used can at least expose what computation takes place. One then studies what possible results can be generated and tries to find situations that are possible but not supported.

## Improperly Constrained Output

Sometimes software developers get all three of the above situations right and then fail to correctly display or transmit responses to users. Such an example can be found in the calendar program shipped with Windows®. An improperly constrained output can be identified by selecting February 29 of a leap year, then incrementing the year. Obviously, February 29 is no longer a valid day. One could legitimately expect the calendar to change the day to either February 28 or, perhaps, March 1. Figure 3 shows the manifestation of this bug: a day that does not exist is displayed. It turns out, however, that this is essentially a screen-refresh problem: selecting the "apply" button will not cause an incorrect date to be stored. The developers simply failed to update the screen.

Finding broken output constraints is similar in nature to finding broken input and data constraints. The task is difficult because it is not always evident from looking at a display panel where the application can trip-up. One has to drive each output to its maximum and minimum value and vary the length and character sets as much as possible and, of course, carefully check each result. Once again, knowledge of the problem domain is important: one has to know, for example, that non-leap years have only 28 days

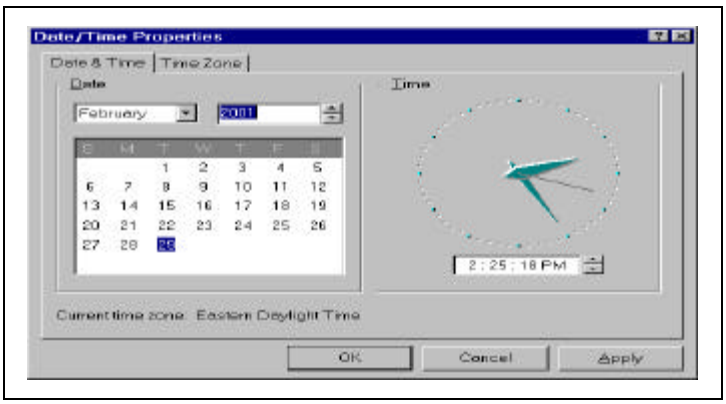



**Figure 3. A Broken Output Constraint**

in February.

## Conclusion and Future Work

This note summarizes a systematic approach for breaking software. Using this approach, we have exposed failures in software that has seen decades of field use and software that has been critically verified by very capable software engineers.

The actual classification of a fault as an input constraint, stored data constraint, computation constraint or output constraint is not particularly crucial. If one interprets a specific fault as a computation constraint instead of a stored data constraint, it makes little difference: a fault is a fault no matter how it is located. Instead, our classification scheme is presented as a way for testers to think through the situations that need testing. It is a guide that testers use to systematically check possible problem areas.

Our current work is focusing on two specific areas concerning our "theory" of software failure: exploitation and remedy. From a testing perspective, knowledge of improperly constrained input, data, computation and output allows us to methodically examine a software product to identify potential faults. From a development perspective, the problem is to design preventative measures directly into software development practices. Our work continues in both of these areas.

## Acknowledgements

Finding bugs in published code and retail applications was the first homework in a Software Testing Methods course taught at Florida Tech in Spring 1999 by the authors. Students in that course gave us many good examples from which to choose for this paper. Special thanks to Steven Atkin, Luis Rivera and Ahmed Stewart whose work appears above. We also gratefully acknowledge the encouragement and participation we received from developers at Microsoft Corporation.

## References

[1] Kernighan, B. and D. Ritchie (1988): *The C Programming Language*, Prentice-Hall.

[2] Mills, H., R. Linger and A. Hevner (1986): *Principles of Information Systems Analysis and Design*, Academic Press.

# JAMES WHITTAKER

Dr. James Whittaker is an associate professor of computer science and the chair of the software engineering program at Florida Tech. He received his Ph.D. from the University of Tennessee in 1992 and regularly consults for major software companies in the United States and Europe.