

An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors*

Anup K. Ghosh & Matthew Schmid
Reliable Software Technologies Corporation
21351 Ridgeway Circle, #400, Dulles, VA 20166
phone: (703) 404-9293, fax: (703) 404-9295
contact email: anup.ghosh@computer.org
www.rstcorp.com

Keywords: software wrappers, testing, robustness, COTS software, fault injection.

Abstract

One of the least tested but most critical portions of software systems is error and exception handling. Error/exception handling routines are the safety net for any system to handle unexpected circumstances such as when operating system (OS) or hardware failures occur. As more critical systems are developed from commercial off the shelf (COTS) software, the robustness of these applications to operating system failures, and in general, to failures from third party software, becomes increasingly critical. In this paper, we present an approach and tool for assessing the robustness of COTS applications to failures from OS functions or other third-party COTS software. The approach consists of wrapping executable application software with an instrumentation layer that can capture, record, perturb, and question all interactions with the operating system. The wrapper is used to return error codes and exceptions from calls to operating system functions. The effect of the failure from the OS call is then assessed. If the application crashes under these anomalous conditions, the application is determined to be non-robust to a particular failing OS call. A failure simulation tool has been developed for testing the robustness of Win32 applications to these types of anomalous OS conditions.

*This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) under Contract F30602-97-C-0117 THE VIEWS AND CONCLUSIONS CONTAINED IN THIS DOCUMENT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL POLICIES, EITHER EXPRESSED OR IMPLIED, OF THE DEFENSE ADVANCED RESEARCH PROJECTS AGENCY OR THE U.S. GOVERNMENT.

1 Introduction

Commercial off-the-shelf (COTS) software is being used increasingly in developing mission-critical systems. For the purposes of this paper, COTS software is any executable software for which source code is not provided or available for analysis. However, in general, off-the-shelf software is any software that is not developed in-house. The time and expense of developing software in-house has spurred developers of critical systems in the transportation, medical devices, and nuclear industries to adopt COTS software in the development of their critical systems. More recently, the Windows 32-bit (Win32) platform, which includes Windows 95/NT/2000/CE operating systems, is being used in mission critical applications.

For example, the U.S. Navy requires that its ships migrate to Windows NT workstations and servers under the Information Technology in the 21st century (IT-21) directive [3]. While modernizing the fleet's technology base is appropriate, the risks of migrating to new platforms are great, particularly in mission-critical systems. One widely-publicized early casualty of this directive involved the USS Yorktown, a U.S. Navy Aegis missile cruiser. The cruiser suffered a significant software problem in the Windows NT systems that control the ship's propulsion system. An application crash resulting from an unhandled exception reportedly caused the ship's propulsion system to fail, requiring the boat to be towed back to the Norfolk Naval Base shipyard [17].

We believe the migration of critical systems to COTS software and to the Windows platform will continue. However, in spite of the headlong rush to adopt COTS software, there exists a dearth of research, technology, and tools for determining the impact of failures of third-party COTS software on the dependability of

the system [18]. That is, the system integrators or maintainers of critical systems have little support to make engineering decisions on what kind of impact the failure of a third party software component will have on their systems, let alone know how to harden their systems for robustness to failures from third party off-the-shelf software. The reality is, no one develops any system, soup to nuts, from custom-built software. Instead, an organization will run its software (or even purchase the application software) on commercial operating systems and use third-party software components to build their applications. In mission-critical systems, it is imperative that the impact of the failure of these third-party components on the application software be known in advance in order to harden the software for robustness.

In this paper, we develop an approach and technology for artificially forcing third-party COTS software to produce exceptions and error conditions when invoked by the software application under study. The goal in developing this technology is to support testing of critical applications under unusual, but known, failure conditions in an application's environment. In particular, we simulate the failure of operating system functions; however, the approach and tool can be used to simulate the failure of other third party software such as imported libraries and software development kits.

The approach is briefly summarized here, then developed in Section 3. Because we are working in the domain of COTS software, we do not assume access to program source code; instead, we work with executable program binaries. The approach is to instrument the interface between the software application under study and the operating system (or third party) software functions the application uses. Fault injection functions are used to simulate the failure of these resources, specifically by throwing exceptions or returning error codes from the third-party functions. The simulated failures are not arbitrary, but rather based on actual observed failures from OS functions determined in our previous study of the Windows NT platform (see [6]), or based on specifications of exceptions and error values that are produced by the function being used. In addition, the approach does not work on models of systems, but on actual system software itself. Therefore, we are not simulating in the traditional sense, but rather forcing certain conditions to occur via fault injection testing that would otherwise be very difficult to obtain in traditional testing of the application. The analysis studies the behavior of the software application under these stressful

conditions and poses the questions: Is the application robust to this type of OS function failure? Does the application crash when presented with this exception or error value? Or does the application handle the anomalous condition gracefully?

In the remainder of this paper, we present some background in robustness testing research, develop the methodology and tool for testing COTS software under these types of stressful conditions, then provide some discussion and conclusions.

2 Background

Robustness testing is now being recognized within the dependability research community as an important part of dependability assessment. To date, robustness testing has focused on different variants of Unix software. B.P. Miller and his research group at the University of Wisconsin wrote a tool called Fuzz that tested many variants of Unix system software to unusual, anomalous, and purely random data [13, 12]. Their results were startling and their paper has been hailed by computer scientists and the dependability research community for shedding light on the quality of commercial OSes versus freeware OSes. The results found that free software performed much better than commercial OSes according to robustness measures. More recently, a group from Carnegie Mellon University led by Phillip Koopman created a tool called Ballista to test OS functions on variants of Unix [8, 10, 9]. Their work is similar to the Fuzz work, except they test OS function calls rather than user level software and they used combinations of valid and invalid data.

In our previous studies of the Windows NT platform, we analyzed the robustness of Windows NT OS functions to unexpected or anomalous inputs [16, 6]. We developed test harnesses and test data generators for testing OS functions with combinations of valid and anomalous inputs in three core Dynamically Linked Libraries (DLLs) of the Win32 Application Programming Interface (API): USER32.DLL, KERNEL32.DLL, and GDI32.DLL. Results from these studies show non-robust behavior from a large percentage of tested DLL functions. This information is particularly relevant to application developers that use these functions. That is, unless application developers are building in robustness to handle exceptions thrown by these functions, their applications may crash if they use these functions in unexpected ways.

In this paper, we are concerned with testing the robustness of application software — specifically mission-critical applications — that run on the Win32 platform. Unlike nominal testing approaches (see [1, 2, 14, 11, 4]) that focus on function feature testing,

we are concerned with testing the software application under stressful conditions. Robustness testing aims to show the ability, or conversely, the inability, of a program to continue to operate under anomalous input conditions. More formally, the IEEE Standard Glossary of Software Engineering Terminology states that robustness is “the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions”.

Testing an application’s robustness to unusual or stressful conditions is generally the domain of fault injection analysis. To date, fault injection analysis of software has generally required access to source code for instrumentation and mutation (see [19] for an overview of software fault injection). In addition, fault injection analysis to date has been performed on Unix-based systems. We seek to develop technologies that will work on COTS-based systems and for the Win32 platform.

To define the problem domain of this work better: an application is robust when it does not hang, crash, or disrupt the system in the presence of anomalous or invalid inputs, or stressful environmental conditions. Applications can be vulnerable to non-robust behavior from the operating system. For example, if an OS function throws an unspecified exception, then an application will have little chance of recovering, unless it is designed specifically to handle unspecified exceptions. As a result, application robustness is compromised by non-robust OS behavior.

We know from our testing of the Win32 platform that the OS functions can throw exceptions and return error values when presented with unusual or ill-formed input. In fact, we know exactly which exceptions and error codes a given OS function will return based on function specifications and our previous experimentation. However, even though this anomalous behavior from the OS is possible (as demonstrated), it is actually unusual during the normal course of events. That is, during normal operation, the OS will rarely behave in this way. Using nominal testing approaches to test the application might take an extremely long time (and a great many test cases) before the OS exhibits this kind of behavior. Thus, testing the robustness of the application to stressful environmental conditions is very difficult using nominal testing approaches.

However, using fault injection functions, we force these unusual conditions from the OS or from other third-party software to occur. Rather than randomly selecting state values to inject, we inject known exception and error conditions. This approach then forces these rare events that can occur to occur. Thus, this

approach enables testing of applications to rare, but real failure modes in the system. The approach does not completely address the problem of covering all failure modes (especially unknown ones), but it does exploit the fact that third-party software does fail in known ways, even if infrequently. At a minimum, a mission-critical application must account for these known failure modes from third-party software. However, many applications never do, because software designers are mostly concerned about the application they are developing and assume the rest of the environment works as advertised. The approach and tool developed here tests the validity of that assumption by forcing anomalous conditions to occur.

It is important to note that we are not necessarily identifying program bugs in the application, but rather we are assessing the ability of the application to handle stressful environmental conditions from third-party software. So, this approach is an off-nominal testing approach that is not a substitute for traditional testing and fault removal techniques. In fact, the approach tests the application’s error and exception handling mechanisms — the safety net for any application. If the application does not account for these types of unusual conditions, chances are that it will fail.

3 Failure Simulation Tool

In order to assess the robustness of COTS-based systems, we instrument the interfaces between the software application and the operating system with a software wrapper. The wrapper simulates the effect of failing system resources, such as memory allocation errors, network failures, file input/output (I/O) problems, as well as the range of exceptions that can be thrown by OS functions when improperly used. The analysis tests the robustness of the application to anomalous and stressful environment conditions. An application is considered robust when it does not hang, crash, or disrupt the system in the presence of anomalous or invalid inputs, or stressful environmental conditions.

From our previous studies of the Windows NT platform, we found a large percentage of OS functions in the three core DLLs of the Win32 API that threw exceptions when presented anomalous input. If these functions are used similarly by an application, then the application must be prepared to handle these exceptions, specified or not. Because testing the application via nominal testing approaches is unlikely to trigger these anomalous OS conditions, we need some alternative method to test the robustness of these applications to OS anomalies without requiring access to

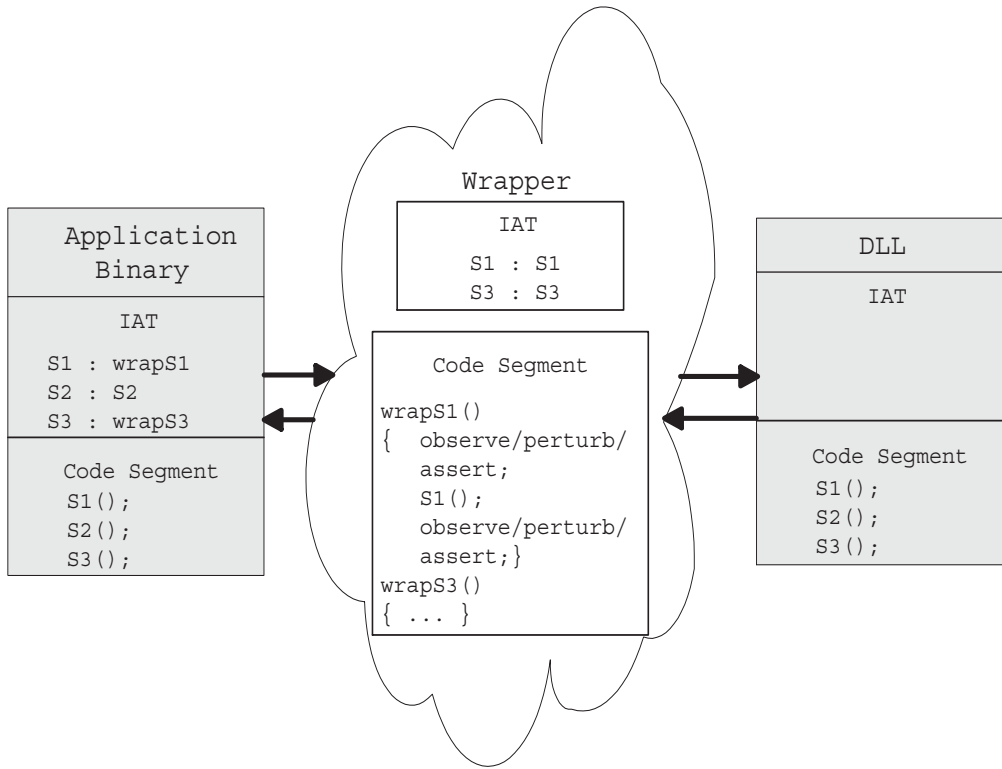


Figure 1: Wrapping Win32 Executable Programs

source code. To address this shortcoming in the state-of-the-art, we have developed the Failure Simulation Tool (FST) for Windows NT.

3.1 Wrapping Win32 Software

The approach that FST employs is to artificially inject an error or exception thrown by an OS function and determine if the application is robust to this type of unusual condition. FST instruments the interface between the application executable and the DLL functions it imports such that all interactions between the application and the operating system can be captured and manipulated.

Figure 1 illustrates how Win32 program executables are wrapped. The application's Import Address Table (IAT), which is used to look up the address of imported DLL functions, is modified for functions that are wrapped to point to the wrapper DLL. For instance, in Figure 1, functions `S1` and `S3` are wrapped by modifying the IAT of the application. When functions `S1` and `S3` are called by the application, the wrapper DLL is called instead. The wrapper DLL, in turn, executes, providing the ability to throw an exception or return an error code to the calling application. In addition, as the figure shows, we also have the abil-

ity to record usage profiles of function calls and the ability to execute assertions.

In practice, the wrapping is accomplished by loading the program under analysis in debug mode [15]. After executing the first instruction, the application is frozen (using our debugger) and the wrapper DLL (shown in Figure 1) is loaded. The wrapper DLL initialization instructions modify the Import Address Table of the application in the manner as shown in Figure 1. That is, for all functions that are to be wrapped, the IAT is modified to point to the modified function addresses. The application is then unfrozen and execution of the wrapped program begins.

There are several ways in which the wrapper can be used. First, the wrapper can be used as a pass-through recorder in which the function call is unchanged, but the function call and its parameters are recorded. This information can be useful in other problem domains such as for performance monitoring and for sandboxing programs. Second, the wrapper can be used to call alternative functions instead of the one requested. This approach can be used to customize COTS software for one's own purposes. For our purposes, we are interested in returning error codes and exceptions for specified function calls. Thus, we develop custom

failure functions for each function we are interested in failing.

Three options for failing function calls are: (1) replace calling parameters with invalid parameters that are known to cause exceptions or error codes, (2) calling the function with the original parameters that are passed by the application, then replacing the returned result with an exception or error code, or (3) intercept the function call with the wrapper as before, but rather than calling the requested function, just returning the exception or error code. Option 3 is attractive in its simplicity. However, options (1) and (2) are attractive for maintaining consistent system state. In some cases, it is desirable to require the requested function to execute with invalid parameters to ensure that side effects from failing function calls are properly executed. Option (1) accounts for this case. Using the information from our previous studies of the Win32 API [6], Option (1) can be implemented by using specifically those input parameters that resulted in OS function exceptions. Alternatively, specifications for which parameters are invalid when using a function (such as one might find in pre-condition assertions) can be used for causing the failure of the function in using Option (1). Option (2) is less rigorous about handling side effects from failing function calls, but will ensure side effects from normal function calls are properly executed. If, however, side effects from calling functions are not a concern, *i.e.*, if the analyst is strictly concerned about how well the application handles exceptions or error codes returned from a function call, then Option 3 is sufficient.

FST modifies the executable program's IAT such that the address of imported DLL functions is replaced with the address to our wrapper functions. This modification occurs in memory rather than on disk, so the program is not changed permanently. The wrapper then makes the call to the intended OS function either with the program's data or with erroneous data. On the return from the OS function, the wrapper has the option to return the values unmodified, to return erroneous values, or to throw exceptions. We use this capability to throw exceptions from functions in the OS (which we found to be non-robust in our earlier studies) called by the program under analysis.

3.2 Obtaining Call Site Coverage

After instrumenting all of the relevant Import Address Table entries, the FST performs a search across the code segment of the module for call sites to the wrapped functions. A call site is identified as a function call instruction followed by a one-word pointer into the IAT. For example, Figure 2 shows the call

sites for LocalAlloc function calls.

This call site information is useful for several reasons. First, it gives the user a sense of how many call sites there are for a particular function. The more instances of a function used, the more critical that function is to the operation of the program. Second, it allows the FST to tally the number of calls on per site basis instead of on a per function basis. That is, the FST allows the analyst to selectively fail certain call sites to a function, while excluding other call sites to the same function from failure. This is particularly useful when attempting to isolate cause and effect of system failure. Finally, and perhaps, most importantly, the call site information gives finer-grained test coverage information. For instance, function coverage is achieved when a given function is called once (by any call site). However, FST provides call site coverage, which not only gives function coverage, but also reveals if a given call site has been executed or not. Call site coverage gives the analyst information on which function call sites have not been exercised, thus facilitating more intelligent testing.

The FST also matches call sites with linked or external debug information, if provided. For instance, if the FST has located a call to HeapAlloc at a specific location in memory, it attempts to determine the corresponding line of source. Though debugging information is not necessarily distributed with an application, it can be a great help to an analyst when it is present. Because of the difficulties involved with juggling the many formats in which debugging information can be distributed, the FST makes use of the Windows NT library ImageHlp.DLL which helps to abstract the details of the debug information format.

Finally, the FST is able to intercept dynamic calls to exported functions. This capability gives FST the ability to instrument functions that are not statically loaded at compile time, but rather dynamically loaded during run-time. The FST provides a special wrapper for the KERNEL32.DLL function GetProcAddress, a function that can be used to retrieve the address of a function at run-time. For functions that the FST would like to wrap, the special GetProcAddress wrapper will return the address of a wrapper function instead of the correct function.

3.3 Using the Failure Simulation Tool

The prototype Failure Simulation Tool provides an interface that helps to simplify the testing of COTS software. There are three primary components to the Failure Simulation Tool: the Graphical User Interface (GUI), the configuration file, and the function wrapper DLLs.

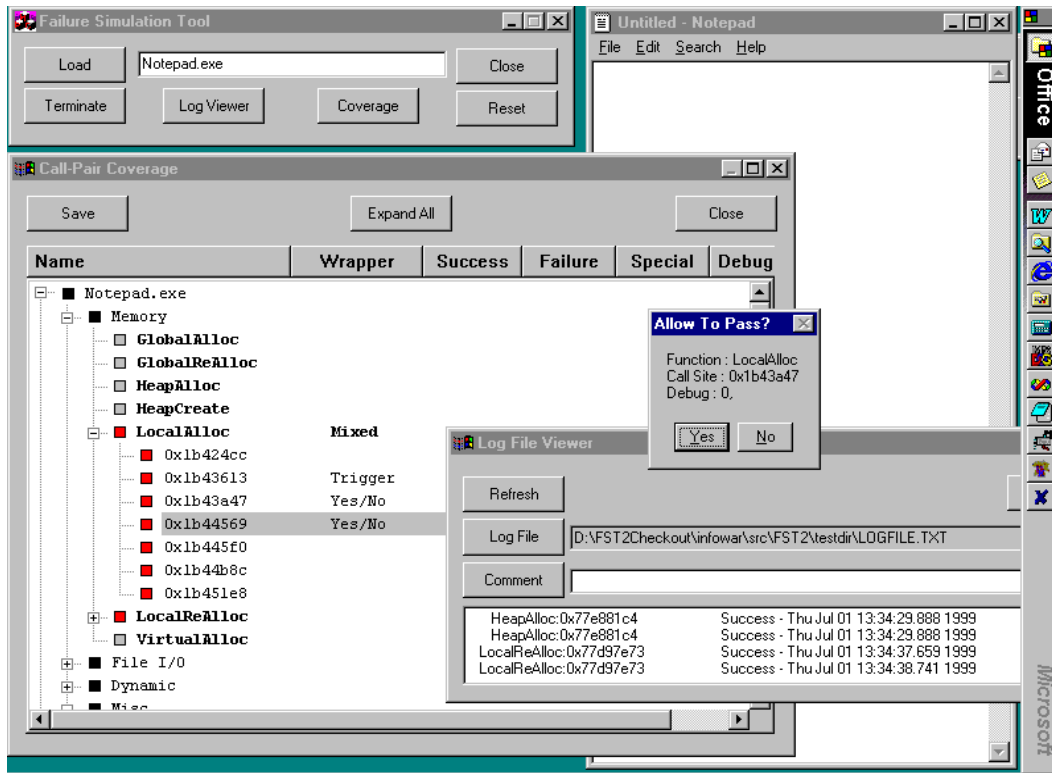


Figure 2: The graphical interface to the Failure Simulation Tool that wraps Win32 executable programs. The tool allows the user to interactively fail OS functions when they are called.

A function wrapper DLL contains the wrappers that will be called in place of the functions that the user chooses to wrap. These wrappers are responsible for simulating the failure of a particular function. The FST includes a variety of wrappers for commonly used functions. Additional wrappers can be developed by a user and used by the Failure Simulation Tool.

The configuration file is used to specify the DLL functions that are going to be wrapped, and the function that is going to be doing the wrapping. The configuration file is also used to break the functions into groups that are displayed by the tree control in the GUI. Here is an example of the configuration file.

```
PAGE: Memory
KERNEL32:HeapAlloc:WRAPDLL:WrapHeapAlloc
KERNEL32:LocalAlloc:WRAPDLL:WrapLocalAlloc
```

This configuration file specifies that there should be a group named Memory, and that there will be two functions in that group. The first function to wrap is HeapAlloc, located in KERNEL32.DLL, and it should be wrapped using the function WrapHeapAlloc, found in WRAPDLL.DLL. The second function is specified in the same manner.

If the application crashes due to exceptions or error values returned by the operating system, then we know that the application is non-robust to exceptions thrown by OS functions that we have shown capable of throwing exceptions.

Figure 2 shows the graphic interface to the FST that allows selective failing of OS resources. The window in the upper left is the Execution Manager. This window is used to select an application for testing, to execute and terminate that application, and to access the other components of the FST. The largest window in Figure 9 is the interface that allows the user to control which functions are currently being wrapped, and what the behavior of those wrappers should be. The window titled "Log File Viewer" maintains a time-stamped listing of all of the function calls that have been made. The small window in the foreground is an example of a wrapper that allows the user to select whether a function should fail or not on a case-by-case basis. In the upper right of this figure is the application being tested - in this case the Notepad.exe application. For a more detailed description of how to use the Failure Simulation Tool, please see the documentation supplied as part of the System User's Manual.

In its current version, the tool is used to interactively fail system resources during execution. The dynamic binding to functions allows the tool to fail or succeed calls to OS functions on the fly in real time. The FST can be used to wrap any Win32 application (mission critical or not) in order to interactively fail system resources and to determine the impact of this failure. The tool is to be used for off-line analysis, in order to determine the effect of system failures prior to deployment. Thus, performance overhead in the analysis is not a large concern, unless it were to introduce unacceptable delays in testing.

4 Discussion

Error and exception handling are critical functions in any application. In fact, error and exception handling make up a significant percentage of the code written in today's applications. However, error and exception handling are rarely tested because: (1) programmers tend to assume well-behaved functionality from the environment, and (2) it is difficult to create these kinds of anomalous behaviors using nominal testing techniques.

In [7], Howell argues that error handling is one of the most crucial, but most often overlooked aspect of critical system design and analysis. For example, Howell cites four examples of the criticality of error handling [7]:

- an analysis of software defects by Hewlett-Packard's Scientific Instruments Division determined that error checking code was the third most frequent cause of defects in their software;
- in a case study of a fault-tolerant electronic switching system, it was found that 2 out of 3 system failures were due to problems in the error handling code;
- many of the safety-critical failures found in the final checks of the space shuttle avionics system were found to be associated with the exception handling and redundancy management software;
- problems with the use of Ada exceptions were a key part of the loss of the first Ariane-5 rocket.

Errors and exceptions are often used by vendors of third-party software components to signal when a resource request has failed or when a resource is being improperly used. For example, exceptions may be thrown when invalid parameters are sent to a function call, or when the requesting software does not have the appropriate permission to request the resource.

Error codes are returned by a function call when an error occurs during the execution of the function. For example, if a memory allocation function is unable to allocate memory, it may return an invalid pointer. Error codes are graceful exits from a function that also allow a programmer to debug the source of the problem. Exceptions can be thrown for similar reasons, but more often, exceptions are thrown for more severe problems such as hardware failures or for cases when it is not clear why a resource request failed. For example, an exception returned by a function call may have actually originated from another function that was called by the requested function. If an exception is not handled by one function, it is passed up the function call chain repeatedly until either some function handles the exception or the application crashes.

In using third-party software, the application developer must be aware of what exceptions can be thrown by the third-party function. Third-party functions can be embedded in libraries such as the C run-time library, software development kits, commercial software APIs, or as part of the core operating system. In most cases, the source code to the third-party function is not available, but header files and API specifications are. The function API, header files, or documentation should declare what exceptions, if any, can be thrown by the third-party function (and under what circumstances). If the application developer does not write exception handlers for these specified cases, then the robustness or survivability of the application is placed at risk if an exception is thrown in practice.

The Failure Simulation Tool is designed to analyze the impact of third party software failures for a given application. The tool allows the analyst to observe the effect of errors or exceptions returned from the OS on the application under analysis. If the program fails to handle exceptions thrown by an OS function it will usually crash.

Currently, a limitation of the tool is its coarse-grained ability to monitor the effects of the fault injection on the target application. Our measure for robustness is crude: an application should not hang, crash, or disrupt the system in the presence of the failure conditions we force. However, it is possible that the third party failures we introduce slowly corrupt the program state (including memory and program registers) that remain latent until a later period after the testing has ceased. It is also possible that while the failure does not crash the program it could simply cause the incorrect execution of the program's functions. Neither of these cases is analyzed by our tool.

To address the former problem, an extensive test-

ing suite would be necessary to gain confidence that even after the failure was caused, the program remains robust. To address the latter problem, an oracle of correct behavior is necessary and a regression test suite would be required after the failure was forced in order to determine correctness of the program. In both of these cases, our tool would falsely label the program as robust to the failure of the third-party component, when in fact the failure introduced a latent error in the application, or the program's output is corrupted without affecting the execution capability of the program. Hence, the scope of our monitoring is limited to the ability of the program to continue to execute in the presence of third-party failures. It does not have the ability to judge the correctness of the functions computed by the application.

5 Conclusions

This paper presents an approach and tool for assessing the robustness of Win32 applications in the face of operating system anomalies. Two factors have motivated this work: first, more and more critical systems are being employed on the Win32 platforms such as Windows NT/95/CE/2000; second, the error/exception handling routines of software applications are rarely tested, but form the critical safety net for software applications. In addition, because most COTS software (such as third-party libraries or OS software) rarely provides access to source code, we constrain our approach to analyzing software in executable format.

Recognizing the vulnerability of software to failing third-party components is simply the first step to hardening the software for robustness. As an offshoot of this work, we developed software wrappers to harden COTS software [5]. The idea is to use knowledge of the vulnerability of software to some exception or error code returned by a third-party component in order to develop a robustness wrapper to handle the anomalous condition on behalf of the application. The technique can determine *a priori* whether this approach will increase its survivability.

In the example of the USS Yorktown, the approach described in this paper can be used to wrap the ship's propulsion system software in order to assess how robust it is to exceptions thrown by the OS. For instance, when a divide-by-zero exception is thrown, the analysis would show that the propulsion system will crash. This information can then be used in turn to develop wrappers to prevent the zero parameter from being passed to the denominator of the divide function, or to handle the divide-by-zero exception gracefully. The most pressing question now for this smart ship is what

other exceptions is the ship's propulsion system and other critical systems non-robust to?

Acknowledgment

Matt Schmid and Frank Hill of RST deserve special recognition for development of the Failure Simulation Tool. Please contact the author if you are interested in using the tool.

References

- [1] B. Beizer. *Software Testing Techniques*. Electrical Engineering/Computer Science and Engineering. Van Nostrand Reinhold, 1983.
- [2] B. Beizer. *Black Box Testing*. Wiley, New York, 1995.
- [3] M. Binderberger. Re: Navy turns to off-the-shelf PCs to power ships (risks-19.75). *RISKS Digest*, 19(76), May 25 1998.
- [4] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10:438–444, July 1984.
- [5] A.K. Ghosh, M. Schmid, and F. Hill. Wrapping Windows NT software for robustness. In *Proceedings of the 29th International Fault Tolerant Computing Symposium (FTCS-29)*, pages 344–347. IEEE Computer Society, IEEE Computer Society Press, June 15-18 1999.
- [6] A.K. Ghosh, M. Schmid, and V. Shah. Testing the robustness of Windows NT software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE'98)*, pages 231–235, Los Alamitos, CA, November 1998. IEEE Computer Society.
- [7] C. Howell. Error handling: When bad things happen to good infrastructures. In *Proceedings of the 2nd Annual Information Survivability Workshop*, pages 89–92, November 1998. Orlando, FL.
- [8] P. Koopman and J. DeVale. Comparing the robustness of posix operating systems. In *Proceedings of the 29th International Fault Tolerant Computing Symposium (FTCS-29)*, pages 30–37. IEEE Computer Society, IEEE Computer Society Press, June 15-18 1999.
- [9] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, pages 72–79, October 1997.

- [10] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the Fault Tolerant Computing Symposium*, June 23-25 1998.
- [11] B. Marick. *The Craft of Software Testing*. Prentice-Hall, 1995.
- [12] B.P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32-44, December 1990.
- [13] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.
- [14] G. Myers. *The Art of Software Testing*. Wiley, 1979.
- [15] M. Pietrek. *Windows 95 System Programming Secrets*. IDG Books, Foster City, CA, USA, 1st edition, 1995.
- [16] M. Schmid and F. Hill. Data generation techniques for automated software robustness testing. In *Proceedings of the International Conference on Testing Computer Software*, 1999.
- [17] G. Slabodkin. Software glitches leave Navy smart ship dead in the water, July 13 1998. Available online: www.gcn.com/gcn/1998/July13/cov2.htm.
- [18] J.M. Voas. Cots: The economical choice? *IEEE Software*, 15(2), March/April 1998.
- [19] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1998.