

Wrapping Windows NT Software for Robustness*

Anup K. Ghosh[†], Matt Schmid, & Frank Hill
Reliable Software Technologies Corporation
21515 Ridgetop Circle, #250, Sterling, VA 20166
phone: (703) 404-9293, fax: (703) 404-9295
email: aghosh@rstcorp.com
www.rstcorp.com

Abstract

As Windows NT workstations become more entrenched in enterprise-critical and even mission-critical applications, the dependability of the Windows 32-bit (Win32) platform is becoming critical. To date, studies on the robustness of system software have focused on Unix-based systems. This paper describes an approach to assessing the robustness for Win32 software and providing robustness wrappers for third party commercial off-the-shelf (COTS) software. The robustness of Win32 applications to failing operating system (OS) functions is assessed by using fault injection techniques at the interface between the application and the operating system. Finally, software wrappers are developed to handle OS failures gracefully in order to mitigate catastrophic application failures.

1 Introduction

Windows NT is rapidly becoming the development, engineering, and enterprise platform of choice. As more and more Windows NT workstations are employed in enterprise-critical and even mission-critical applications, the dependability of the Win32 platform is becoming increasingly important. For example, the U.S. Navy requires its ships to migrate to Windows NT workstations and servers under the Information Technology in the 21st century (IT-21) directive [3]. One casualty of this policy has been the USS Yorktown, a U.S. Navy Aegis missile cruiser, which suffered a significant software problem in the Windows NT systems that control the “smart ship”. Reportedly, a database overflow error resulting from a divide

by zero operation caused the ship’s propulsion system to fail leaving the ship effectively dead in the water [13]. The ship had to be towed to the Norfolk Naval shipyard.

Despite the proliferation of NT Workstations in enterprise- and sometimes mission-critical environments, little analysis of the software that composes the NT platform has been performed. As a result, the decision to employ Win32-based systems (Windows 95/NT/2000/CE) in critical applications is based on anecdotal evidence or trust in the software vendor rather than based on scientific study.

To date, analysis on the reliability of operating systems has been performed for commercial and free software variants of Unix [7, 6, 10, 9]. The purpose for studying the reliability of the operating system (OS) and its associated software is to determine the extent to which confidence can be placed in the platform for running enterprise- and mission-critical applications. Once non-robustness in OS functions is identified, either the application software itself can be hardened, or software wrappers can be written for COTS software to gracefully handle non-robust OS failures. The latter approach is developed in this paper.

Robustness testing is now being recognized within the dependability research community as an important part of dependability assessment. Unlike traditional testing approaches [1, 2, 11, 8, 4] that either attempt to find errors through debugging tests, or show reliability through sampling the operational profile, 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”. For the purposes of this work, an

*This work is sponsored under the Defense Advanced Research Projects Agency (DARPA) 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.

[†]Ghosh is the correspondence author.

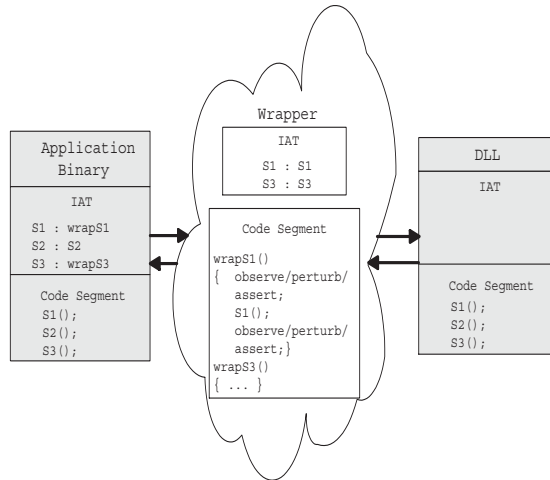


Figure 1: Wrapping Executable Binaries

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.

In this paper we describe an approach for assessing then improving the robustness of the Win32 application software to anomalous and unexpected events. In our previous research on the Win32 platform, we identified a number of OS functions in the core dynamic link libraries (DLLs) of the Win32 API that throw exceptions when presented anomalous input [12]. Here, we use fault injection functions to artificially throw these exceptions in order to test the robustness of applications under these conditions. When applications fail to handle exceptions thrown by the operating system, they crash. In order to address this non-robustness, we developed a wrapping technology to handle anomalous behavior from the operating system on behalf of the application. These wrappers can be applied to COTS software applications that behave non-robustly to failing OS functions.

2 Fault Injection for Testing Application Robustness

From our previous studies of the Win32 platform [12, 5], we know not only which functions behave non-robustly, but also the specific input that results in exceptions being thrown by the operating system. This information can in turn be used to determine the robustness of software applications to exceptions thrown by these functions.

Rather than test the application until the unusual operating system condition occurs (an indefinite period of testing), our approach is to artificially inject anomalous behavior from an OS function and deter-

mine if the application is robust to the exception. The approach involves instrumenting 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 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 modify, perturb, question or simply log the request to the target DLL.

A failure simulation tool has been written that modifies the executable program's import address table 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 called by the program under analysis. If the application crashes due to these exceptions thrown by the operating system, then we know that the application is non-robust to exceptions thrown by OS func-

The wrapper approach is particularly useful for mission-critical COTS software, where access to the source code is not available, but where robustness is important. The wrapper can be deployed with the application such that whenever the application is started, it is started with the wrapper in place. The degradation in performance due to the wrapper is not discernible from a user perspective. As a proof of concept, we wrapped Microsoft Word with a protective wrapper that converts exceptions to error codes. Using the wrapper, we sent in a zero as the first parameter to the `HeapAlloc()` function, which we know will cause an exception to be thrown. The protective wrapper trapped the exception thrown by `HeapAlloc()` and returned an error value (a null pointer). Rather than crashing, Word continued to function by attempting to allocate memory again. In our test case, we sent the erroneous parameter only once and Word recovered nicely.

4 Conclusions

This paper presented an approach to assessing the robustness of Win32 applications to non-robust OS functions, and an approach to hardening the application with a protective wrapper in order to provide application robustness.

The approach uses software wrappers to test robustness of applications in the presence of non-robust OS functions. A failure simulation tool was written to allow selective failures from OS functions in the form of thrown exceptions. If the program fails to handle the exception thrown by an OS function it will crash, rendering the system non-robust and unreliable.

The second part of this paper developed an approach to writing software robustness wrappers to protect applications from failing OS functions. The technique uses the software wrapping approach to handle exceptions thrown by the OS by returning specified error codes or other exceptions that are known *a priori* to be handled robustly. This technique is particularly useful for COTS software where access to source code is not made available and where robustness is critical.

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. Ghosh, M. Schmid, and V. Shah. Testing the robustness of windows nt software. To appear, November 4-7 1998.
- [6] 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.
- [7] 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.
- [8] B. Marick. *The Craft of Software Testing*. Prentice-Hall, 1995.
- [9] 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.
- [10] 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.
- [11] G. Myers. *The Art of Software Testing*. Wiley, 1979.
- [12] M. Schmid and F. Hill. Data generation techniques for automated software robustness testing. In *Proceedings of the International Conference on Testing Computer Software*, 1999. To appear.
- [13] 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.