

Why COTS Software Increases Security Risks*

Gary McGraw

Reliable Software Technologies

21515 Ridgetop Circle, Suite 250, Sterling, VA 20166

phone: (703) 404-9293, fax: (703) 404-9295

email: gem@rstcorp.com

http://www.rstcorp.com

Abstract

Understanding the risks inherent in using COTS software is important because information systems today are being built from ever greater amounts of reused and pre-packaged code. Security analysis of complex software systems has always been a serious challenge with many open research issues. Unfortunately, COTS software serves only to complicate matters. Often, code that is acquired from a vendor is delivered in executable form with no source code, making some traditional analyses impossible. The upshot is that relying on today's COTS systems to ensure security is a risky proposition, especially when such systems are meant to work over the Internet. This short paper touches on the risks inherent some of today's more popular COTS systems, including Operating Systems and Java Virtual Machines. I also present robustness results gathered by a research prototype called RIDDLE (Random and Intelligent Data Design Library Environment), which was used to assess the robustness of native Windows NT system utilities as well as Win32 ports of the GNU utilities.

1 COTS in Action (or, COTS Inaction)

Like the rest of the Department of Defense, the United States Navy is mandated to use Commercial Off-The-Shelf (COTS) technology in order to standardize and to save money. The Navy's *Smart Ship* initiative, which is currently being tested as a pilot study on the Aegis missile cruiser USS Yorktown, is a prime example of the move to COTS. A major part

*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.

of the initiative is to migrate systems to the Microsoft Windows NT Operating System. What recently happened to the Yorktown serves to underscore the nature of security risks inherent in COTS-based systems.

In September 1997, the Yorktown was underway in maneuvers off the Virginia coast. During the maneuvers, the Yorktown suffered a serious systems failure caused by a divide by zero error in an NT application. According to the RISKS digest (Volume 18, Issue 88), the zero seems to have been an erroneous data item entered by a system user. As a result of the error, the ship was dead in the water for over two and a half hours.

This somewhat amusing anecdote would turn out to be a very serious and potentially deadly problem during wartime. Windows NT is known to have a number of failure modes, any one of which could be leveraged into an Information Warfare weapon. Nevertheless, since NT is quickly becoming a *de facto* standard in industry, the DoD is unlikely to abandon its effort to adopt it. Instead of becoming less likely, problems such as those experienced on the Yorktown are a hint of things to come.

2 COTS Problems Percolate Up

Despite the proliferation of NT Workstations in business-critical and mission-critical environments, little analysis of the software that comprises the NT platform has been performed. This implies that the extent to which NT has inherent security risks, systems built with a COTS architecture that include NT inherit the same risks.

Operating Systems are not alone in this problem. Any third-party software included in a system has the same risk-percolation property, whether the software is packaged at the component level or higher. That means that COTS parts of electronic commerce systems now on the drawing board, including Web

browsers and Java Virtual Machines, introduce similar concerns [2].

If COTS introduce more risks, why use them? Unfortunately, the issue is not completely cut-and-dry. COTS have a number of important benefits that should not be overlooked. Consider component re-use in programming systems like Visual Basic. Today's applications are more complicated than ever, and the time pressure to get them done and put them into use is greater than ever. Visual Basic components save both time and effort. There is no reason that the software industry should not learn from electrical engineering where prefabricated components have been used for years.

The real problem is this:

COTS often suffer from dependability, security, and safety problems. What can we do to analyze COTS and measure them according to these properties?

This problem is exacerbated by the fact that COTS are usually delivered with no guarantees about their behavior in *black box* form. It is hard enough to try to determine what a program will do given its source code. Without the source code, the problem becomes much harder.

3 Risks in Java: A case study

The Java programming language from Sun Microsystems makes an interesting case study from a COTS security perspective. Java was, after all, designed with security in mind. Java's security mechanisms are built on a foundation of type safety and include a number of language-based enforcement mechanisms[4]. Unfortunately, as with any complex system, Java has had its problems with security. It turns out to be very hard to do things exactly right, and exactly right is what is demanded by security.

Ed Felten and I have defined four broad categories of attacks with which to understand Java's security risks. These categories can be used to categorize all mobile code risks:

1. **System modification attacks** occur when an attacker exploits a security hole to take over and modify the target system. These attacks are very serious and can be used for any number of nefarious ends, including: installing a virus, installing a trap door, installing a listening post, reading private data, and so on.
2. **Invasion of privacy attacks** happen when a piece of Java code gets access to data meant to be

kept private. Such data includes password files, personal directory names, and the like.

3. **Denial of service attacks** make it impossible to use a machine for legitimate activities. These kinds of attacks are almost trivial in today's systems (Java or otherwise) and are an essential risk category for e-commerce and defense.
4. **Antagonism attacks** are meant to harass or annoy a legitimate user. These attacks might include displaying obscene pictures or playing sound files forever.

Unfortunately, all four categories of attack can be carried out in Java systems. By far the most dangerous attacks, system modification, leverage holes in the Java Virtual Machine to work. Though Java's internal defenses against such attacks are strong, at least fifteen major security holes have been discovered in Java (and since patched). The latest such hole, a problem with Class Loading in the JDK 1.2beta3, was discovered in July 1998.

If supposedly-secure systems like Java Virtual Machines (items commonly included as COTS in systems ranging from smart cards and embedded devices to Web browsers) have security risks, what does this say about less security-conscious COTS? The somewhat disturbing answer is that other systems are much worse off. Microsoft's ActiveX system, for example, presents a number of far more serious security problems than Java does.

4 Black Box Analysis

Most software security vulnerabilities result from two factors: program bugs and malicious misuse. Technologies and methodologies for analyzing software in order to discover these vulnerabilities (and potential avenues for exploitation) are a current topic of computer security research. Dynamic software analysis technologies usually require program source code. However, most COTS software applications are delivered in the form of binary executables (including hooks to dynamic libraries), rendering source-code-based techniques useless. Thus, alternative methods for analyzing software vulnerability under malicious misuse or attack are required.

Dynamic black-box analysis is an important approach to software vulnerability localization that, given today's inexpensive hardware, can be performed relatively cheaply. This analysis is a variant on traditional software testing that is particularly attractive because it can be applied to binary executables, including COTS and *legacy* executables. This approach

is not typical vanilla testing, but rather focused testing with the express purpose of determining a component’s tolerance to attack. Though this approach neither requires functional specifications for components nor functional requirements, it does require the user to characterize what a security violation is (based on site-specific security policy).

4.1 RIDDLE: NT robustness

Research at Reliable Software Technologies is addressing the problem of COTS security analysis by beginning with the problem of component robustness [1]. This follows the footsteps of two research efforts: Fuzz [5] and Ballista [3], both of which considered the robustness of Unix system software. RIDDLE’s target is NT.

The Random and Intelligent Data Design Library Environment (RIDDLE) enables analysis of commercial off-the-shelf (COTS) software by using black-box testing techniques. RIDDLE permits stress testing of application software, system utilities, COM/DCOM components, shared libraries, and system functions. Unlike traditional black-box testing approaches, RIDDLE stress tests software using unexpected, intelligently crafted test cases. The goal of this research is to determine what robustness gaps, if any, exist in Windows NT software.

Test cases are generated with random, intelligent input using the input grammar of the component under analysis. Rather than simply generating random input that does not meet the basic syntax of the program’s input, generating input intelligently using the input grammar of the component permits stress testing of more of the software’s functions. RIDDLE provides an environment to combine random input, malicious input, and boundary value conditions in the legal grammar of the program to test its behavior more thoroughly under anomalous conditions.

4.2 Results

RIDDLE was used to perform robustness tests on two categories of Windows NT software. The first category is made up of Windows NT command line utilities that are supplied with the operating system. The utilities tested are `attrib`, `chkdsk`, `comp`, `expand`, `fc`, `find`, `help`, `label`, and `replace`. The second category of software that was tested was a group of GNU command line utilities that have been ported to the Windows NT operating system as part of the Cygnus GNU-Win32 project. The ported GNU utilities tested are `cat`, `chmod`, `chksum`, `cp`, `ls`, `mv`, `rm`, and `wc`.

The experimentation covered many combinations of the string lengths and character sets. In all, there were 64,000 tests run on the GNU utilities, and 114,000

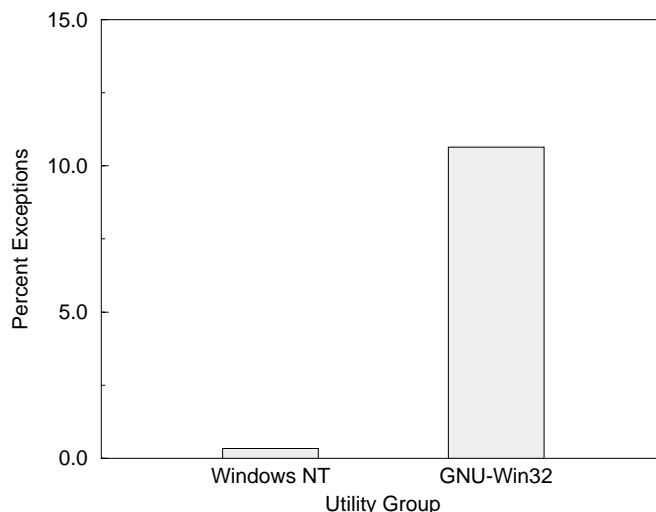


Figure 1: Percentage of unhandled exceptions for all test cases run against the native Windows NT and GNU-Win32 utilities. The vast majority of unhandled exceptions were memory access violations that result in the aborted execution of the program being tested.

tests run on the native Windows NT utilities. RIDDLE detected distinct termination states from the programs that were tested. The exit states determine when the program terminates normally, when the program is hung, and when the program terminates due to an unhandled exception. Three types of exceptions were caught by the RIDDLE monitor in these experiments: memory access violation exceptions, privileged instruction exceptions, and illegal instruction exceptions. If these exceptions arise during the execution of a program, then the program has failed to perform robustly by failing to handle the exception internally.

Figure 1 summarizes the results of the testing of native Windows NT utilities and the GNU-Win32 utilities. In all the test cases run against the native Windows NT utilities, only 0.338% of the test cases resulted in failure according to our failure metric. On the other hand, the GNU-Win32 utilities exit with an unhandled exception 10.64% of the time. The distribution of exceptions favored memory access violations so heavily (approximately 7000 to 1 for GNU-Win32, and 100 to 1 for Windows NT) that the other types of exceptions are statistically insignificant.

Further analysis of the GNU-Win32 results show that the 10.64% failure rate is fairly consistent across the eight GNU utilities that were tested. The vast majority of the exceptions occurred when the char-

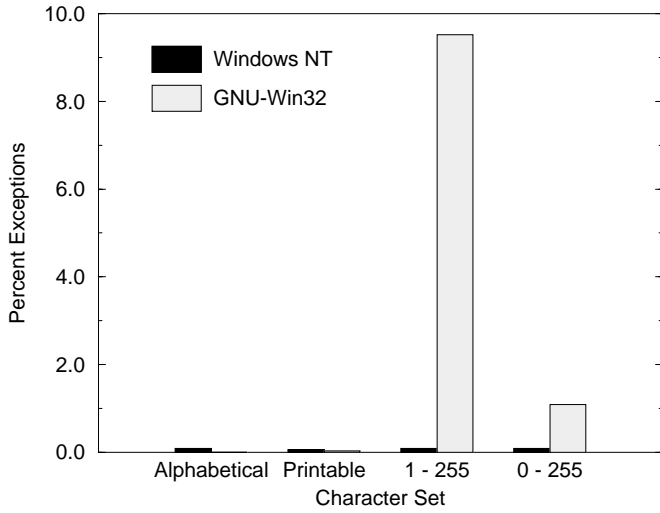


Figure 2: Distribution of unhandled exceptions among different content types. The character range $[1,255]$ includes all characters except the NULL character. The range of the last column, $[0,255]$, includes the NULL character.

acter set being used for string generation was in the range $[1,255]$ (excluding the NULL character). This is most likely due to the program’s interpretation of special characters. The number of exceptions decreases dramatically when the character set is altered to include the null character, or when it consists only of printable characters in the range $[33,127]$. The null character may be interpreted as the termination character of a string, effectively limiting the length of the input. This would explain why there are fewer unhandled exceptions when this character is used in light of the correlation between length and exceptions (see Figure 3). Another possibility is that if the null character is interpreted as either the end of a string or the end of the parameter list, then the parameters may no longer constitute a valid use of the application and the utility may immediately reject the test case.

The distribution of exceptions by content type is shown in Figure 2. Clearly, the GNU-Win32 utilities are most vulnerable to input that is sampled from the character set range $[1,255]$. This set includes every printable and non-printable character except for the NULL character. Even very long length input in the alphabetical and printable set resulted in few exceptions. Instead, it is the combination of very long length with nearly the entire range of the character set (including non-printable characters) that resulted

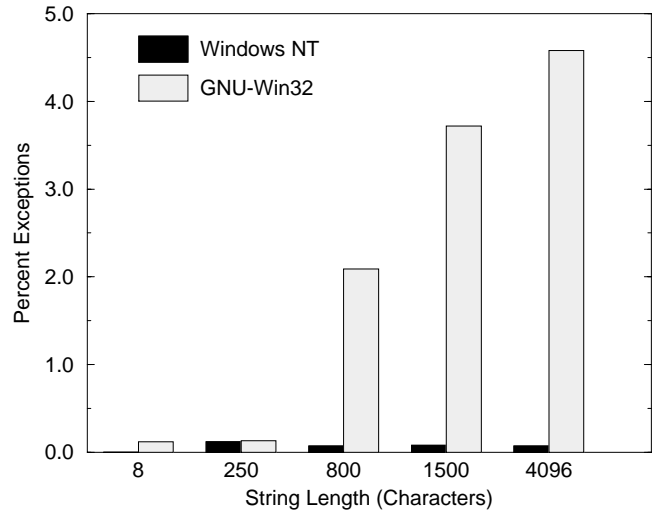


Figure 3: The percentage of test cases that resulted in exceptions as a function of the length of the input strings.

in the most unhandled exceptions.

The most significant trend in the data collected from the tests performed on the GNU-Win32 utilities is the increase in unhandled exceptions as the length of string increases as illustrated in Figure 3. The graph of the GNU-Win32 exception ratios show that as the length of input is increased from 8 to 4096 bytes, the number of exceptions rises dramatically indicating a failure to handle anomalous input within proper input grammar. Significantly fewer exceptions occurred when the length of the string used was either 8 or 250 characters. Because the exception that occurred most often was a memory access violation, the cause is most likely an over-written buffer that placed an illegal pointer on the program stack. In other words, the instruction pointer that was overwritten with the long input probably points to a region of the memory that is inaccessible for the program, or it may point to data that is not a valid instruction opcode. This result points to potential vulnerabilities in the GNU utilities to buffer overrun attacks.

The data collected from the tests run on the native Windows NT utilities paints a very different picture. Of the nine utilities that were tested, only two of them, `comp` and `expand` produced any exceptions. The `expand` utility had a failure pattern similar to the GNU-Win32 utilities. It failed more often when the strings were longer and the character sets were more complex. The `comp` utility failed most frequently when

the character set was alphabetic, and the string length was 250.

5 Towards Managing COTS Risks

It is clear that much work remains to be done in understanding the security implications of using COTS. COTS are becoming as ubiquitous as software itself. Given that COTS specifically designed with security in mind (like the Java VM) suffer from serious security problems, we can only cringe at the thought of the risks that less carefully-designed COTS introduce.

The RIDDLE experiments show that probing the behavior of a black box COTS system is a useful exercise. RIDDLE is currently being expanded to support testing of network servers, shared libraries, desktop applications, and OLE/COM/DCOM components. Future research will involve testing these other classes of NT software as well as exploring robustness gaps to determine their potential to be exploited into security holes.

Acknowledgements The RIDDLE work sketched here was performed by Anup Ghosh, Matt Schmid, and Viren Shah of Reliable Software Technologies. See [1] for a more thorough treatment of the subject.

References

- [1] A. Ghosh, M. Schmid, and V. Shah. Testing the robustness of windows nt software. To appear, November 4-7 1998.
- [2] A.K. Ghosh. *E-Commerce Security: Weak Links, Best Defenses*. John Wiley & Sons, New York, NY, 1998. ISBN 0-471-19223-6.
- [3] 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.
- [4] G. McGraw and E. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, New York, 1996.
- [5] 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.