

Why COTS Software Increases Security Risks

Gary McGraw and John Viega

Reliable Software Technologies

21515 Ridgetop Circle, Suite 250, Sterling, VA 20166

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

email: {gem,viega}@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.

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 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 and robustness 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 [1]. Unfortunately, if a vendor embeds COTS software in a product, end users will not absolve the vendor of blame for any system failures.

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 So Why Use COTS?

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. The main benefits of COTS software are the following:

1. **Reduction in development time.** There is often a great demand to produce software as quickly as possible. One driving factor is that many vendors need to stay one step ahead of competing products.
2. **A reduction in lines of code to be written.** COTS components provide functionality the developer would have had to write, otherwise. Thus, the code that would have been required to implement that functionality is saved. An exception is the “glue” code that ties a system to the COTS component. Such code, however, tends to be a small part of a program.
3. **A reduction in complexity faced by the developer.** COTS components can provide abstractions that can hide complexity that a developer would otherwise have to tackle. Not only can this benefit help reduce development time, it can potentially lead to fewer errors in the non-COTS portions of the system.

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.

4 Security 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[6]. 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.

[7] defines 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 sixteen major security holes have been discovered in Java (and since patched). The latest such hole, a problem with class loading in Java 2, was discovered in March 1999.

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.

5 Current Options

We see the following options as available to a vendor who would like to use COTS software:

1. Avoid COTS altogether; write all software in-house.
2. Depend on the law for protection.

3. Demand warranties from COTS vendors.
4. Perform system level testing with the COTS software embedded.
5. Ask for independent product certification, as opposed to personnel or process certification.
6. Determine the robustness of the software if the COTS software were to fail.
7. Use software wrappers.

Avoiding COTS software completely gives the vendor quite a bit of control, but foregoes all benefits associated with COTS, so is not generally desirable. The legal system might provide some respite for the vendor who uses COTS software, but laws covering software are few and far between, and their enforceability is in question, due to unsatisfactory legal precedents. Warranties may be enforceable, but tend to be difficult to obtain, since adequate testing is such a difficult proposition, even with an extremely large user base.

5.1 Software Testing

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. White-box 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. Performing similar analyses at the binary level has not been shown to be feasible. Thus, alternative methods for analyzing software vulnerability under malicious misuse or attack are required.

Black-box analysis is an important approach to software vulnerability localization that, given today's inexpensive hardware, can be performed relatively cheaply. Such analyses are particularly attractive because they can be applied to binary executables, including COTS and *legacy* executables.

One reasonable methodology is using random and intelligent data generation, and then monitoring results based on a security policy, which could be enforced by hand, or by a sandbox. We have experimented applying such a methodology to test the robustness of Windows NT software, with highly encouraging results [4]. That work follows the footsteps of two other research efforts, Fuzz [8] and Ballista [5], both of which considered the robustness of Unix system software.

5.2 Product Certification

There are many certification standards that can be applied to software process, such as ISO 9000, and the Carnegie Mellon Capability Maturity Model. There are also ways to certify the people who build software. Such certification models can not offer high levels of assurance since they do not test the actual software. A good analogy is that dirty water can run from clean pipes; even the best process, carried out by the best people can produce shoddy software.

Currently, there is no tried and true approach for certifying general-purpose software for robustness or security. Developing such a certification process in which people can reasonably place their trust is quite difficult, since tools for detecting such problems tend not to be effective enough in practice. Such tools would need to be able to accurately quantify the robustness or security of a piece of software. One effective approach for quantifying robustness is *fault injection*, which is described in the next subsection.

Recently we have begun to develop a process that can be used to certify for security that is still under development [2].

5.3 Robustness Testing

One way for developer to gain reasonable assurance of the robustness of a COTS component is to test the component in-house. Currently, the best verification and validation technique for doing so is software *fault injection* [9]. Fault injection is a “what-if” analysis technique, and is not statistical testing, or any sort of proof of correctness. Faults are injected into a piece of software, and then the behavior of the modified software is observed for anomalous behavior¹. The intuition behind this technique is that the more “what-if” games you play, the more confident you become that your software can overcome anomalous situations. In order to work well, the injected faults need to be representative of the types of actual problems that are likely to crop up in the program. Fault injection techniques have long been used to test for robustness of both hardware and software, and recently have been applied successfully to the security domain [3].

Fault injection techniques can be tailored to COTS systems by systematically injecting faults into the inputs of the COTS application, and then the outputs of the COTS software is observed for unacceptable results. An overview of such an architecture is shown in Figure 1.

¹Note that it is possible for a COTS component to return an erroneous output without having a negative impact on the system as a whole. Such cases are generally not considered intolerable, but are also uncommon.

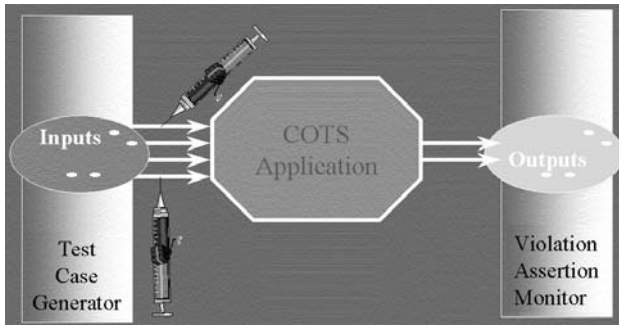


Figure 1: Applying fault injection techniques to test embedded COTS software

5.4 Software Wrapping

Fault injection techniques can be used to identify unacceptable behavior exhibited by a COTS component. What they cannot do is automatically correct those problems. Since the system developer generally does not have access to the source of COTS components, it is also not usually feasible to repair detected faults. One way that can help alleviate the problem is for the developer to “wrap” the COTS software, disallowing it from exhibiting undesired functionality by placing a software barrier around the component, limiting what it can do. Generally, the way in which such wrapping is done is by testing inputs before a call, and only allowing the call if the input is likely to produce acceptable behavior. An overview of this technique is shown in Figure 2.

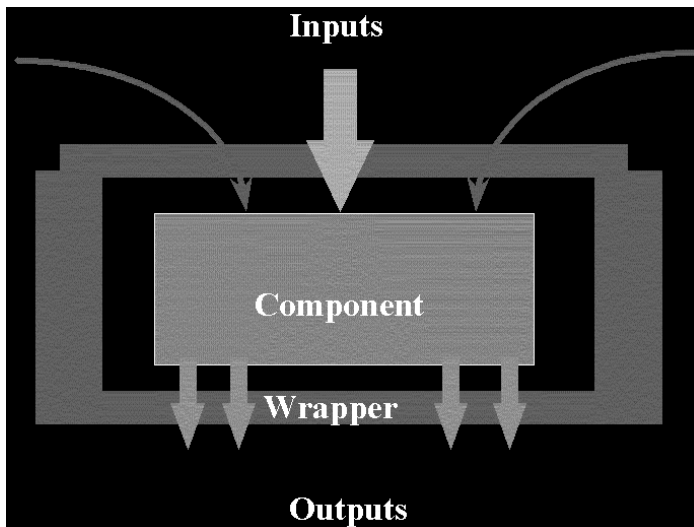


Figure 2: Wrapping a COTS component

This technique is both complex and error prone. One significant problem with the approach is that the

system developer must write code that gracefully handles inputs that can not properly be handled by the COTS component. However, we have yet to encounter a more practical approach to addressing the problem of gracefully avoiding robustness problems in COTS software.

6 Conclusion

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

Using COTS software is a gamble, but it is one that many developers must take. For those developers, we recommend the following course of action:

1. Perform thorough system-level testing.
2. Request warranties from vendors of COTS software.
3. Assess how your system will tolerate COTS failures.
4. Take your own mitigating steps by wrapping COTS components.

References

- [1] A.K. Ghosh. *E-Commerce Security: Weak Links, Best Defenses*. John Wiley & Sons, New York, NY, 1998. ISBN 0-471-19223-6.
- [2] A.K. Ghosh and G. McGraw. An approach for certifying security in software components. In *Proceedings of the National Information Systems Security Conference*, October 6-9 1998. Crystal City, VA USA.
- [3] A.K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 104-114, Oakland, CA, May 3-6 1998.
- [4] 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.

- [5] 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.
- [6] G. McGraw and E. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, New York, 1996.
- [7] G. McGraw and E. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, New York, NY, 2nd edition, 1999.
- [8] 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.
- [9] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1998.