burton GROUP™

Application Security: Everybody's Problem

Version: 1.0, Aug 04, 2005

AUTHOR(S):

Diana Kelley
(dkelley@burtongroup.com)

TECHNOLOGY THREAD:

Application and Content Security

Conclusion

Integrating security into the software development lifecycle (SDLC) will uncover security flaws earlier in the process and should result in more robust, less vulnerable applications. Although there is no replacement for security-aware design and a methodical approach to creating more secure applications, code-scanning tools are a very useful addition to the process. Enterprises that understand how to create more secure applications can benefit from greater efficiencies in the development process, and they may find that by creating more secure applications, there is less need for post-production security software whose entire function is to protect and patch insecure software.

BURTON GROUP 7090 Union Park Center · Suite 200 · Midvale, Utah 84047 P 801.566.2880 F 801.566.3611 www.burtongroup.com

Publishing Information

Burton Group is a research and consulting firm specializing in network and applications infrastructure technologies. Burton works to catalyze change and progress in the network computing industry through interaction with leading vendors and users. Publication headquarters, marketing, and sales offices are located at:

Burton Group
7090 Union Park Center, Suite 200
Midvale, Utah USA 84047-4169
Phone: +1.801.566.2880
Fax: +1.801.566.3611
Toll free in the USA: 800.824.9924
Internet: info@burtongroup.com; www.burtongroup.com

Table Of Contents

Synopsis

Most enterprises rely on commercial off-the-shelf (COTS) and proprietary software applications to manage and conduct business. Software applications run everything from manufacturing and human resources to financial reporting, financial transactions, and sales. The reliability of these applications can directly impact the success of the enterprise. Although many studies1,2 have shown that it is less expensive to fix defects earlier in the development process, many development and design teams do not address security-related defect control until far into the cycle. The application development process is focused on features and functions. The software development lifecycle (SDLC) encompasses quality assurance planning and testing but rarely addresses security in a comprehensive manner. The advent of the need for faster lifecycles to keep pace with "Internet time" and agile development models has not replaced the need for an SDLC.

Creating applications based only on features and functions—without addressing the underlying requirements for robust, reliable, secure code—often results in software that fails the real-world security test in production. This can lead to financial loss and costly cycles of frequent patching and upgrades.

Although a contributing factor, the fundamental problem is not simply that application developers are often inadequately trained in the art of secure coding. The problem is magnified, in large part, because many commercial and proprietary software applications are built to meet a set of low-surety requirements. If the underlying tenets of security—integrity, availability, confidentiality, use control, and accountability—are not incorporated during the requirements phase, the resulting application will probably not have them. Little, if any, trace-back accountability exists for a defect that causes major loss to a customer years into the future.

In the development process, enterprises must decide what levels of risk are acceptable and then create and operate the software development process in a way that meets those needs.

This report examines ways that companies can use tools and techniques to decrease flaws and increase surety levels of their C, C++, and Java applications. The report also discusses why it is important to understand security requirements throughout the development process, starting with the requirements phase, and how companies can integrate more secure requirements, design, and coding practices into the SDLC. Finally, the report will detail available C, C++, and Java code-scanning tools, both source and binary, as well as other processes and approaches that can be implemented to increase the security of applications.

Analysis

In order to create more secure applications, security awareness must be an integral part of the entire software development lifecycle (SDLC). Trying to create secure code using a one-time-event approach is not reasonable. Creating more secure software takes a village, so to speak. Secure development is a concerted, ongoing effort between all the parties involved with designing and creating code to ensure that appropriate security requirements are met. This begs the question: Is more secure code necessarily better? And the answer is: It depends on an enterprise's specific need for the surety level of the application as it functions within the enterprise's environment. There is no one-size-fits-all security level for all applications and all enterprises. There is no such thing as absolute application security. There is, however, an appropriate security level. More information on determining appropriate surety levels for applications can be found in the *Security and Risk Management Strategies* report, "Building Secure Applications: How secure do you want to be today?"

Some applications may require higher levels of surety based on their function within the enterprise. Applications that are highly reliant on information integrity may require higher levels of surety, especially in cases for which failure could result in catastrophic repercussions. For example, if a banking application were coded in such a way that transfers were processed to incorrect accounts, serious financial loss could ensue for both the user and provider of the system. Taking this example further, if the transfers were for a large portion of the bank's total assets, and the transfers were made to accounts outside U.S. jurisdiction, the loss could potentially be large enough to put the bank out of business. While this specific scenario would most likely be blocked by the Federal Reserve Bank's documentation requirements, similar scenarios are certainly feasible and their consequences potentially dire.

A real-world integrity problem was encountered by the crew of the USS Yorktown, a guided missile cruiser. As reported in the November 1998 *Scientific American*, "After a crew member mistakenly entered a zero into the data field of an application, the computer system proceeded to divide another quantity by that zero. The operation caused a buffer overflow, in which data leaked from a temporary storage space in memory, and the error eventually brought down the ship's propulsion system. The result: The Yorktown was dead in the water for more than two hours."3 Had the buffer overflow and resultant failure of the propulsion system occurred when the ship was engaged in active combat, the lives of the crew and control of the ship would have been at risk.

A confidentiality example pertains to applications that are directly involved with processing personally identifiable information (PII), such as storing Social Security numbers (SSNs). Applications that handle data, where the mismanagement of such data could result in medium or serious consequences for the enterprise, often have higher surety-level requirements than applications that do not handle such sensitive data. More examples will be provided later in the report.

The vast majority of today's businesses are dependent on software applications. These applications are relied on to conduct critical functions such as managing the inventory in a channel pipeline, ensuring that the correct parts are on a manufacturing shop floor, and maintaining the integrity of financial information and reporting. Problems with these applications can have serious repercussions. It is of value to remember that preventing attacks and misuse are not the only reason to address the software security and reliability issue. As in the case of the USS Yorktown, software faults in and of themselves can cause failure. Another famous example of this occurred on June 4, 1996, when the unmanned satellite launcher Ariane 5 destructed 40 seconds after take-off. According to the official report,4 the problem was caused by an overflow Operand Error and a failure to properly manage the exception handling caused by this error in the software managing the launcher.

The USS Yorktown and Ariane 5 examples show that software can fail even when no malicious acts are initiated by attackers. But enterprise software is also at risk of malicious attack. Most attackers go for what is commonly referred to as the low-hanging fruit (the most easily exploited vulnerabilities) when identifying attack targets and vectors. In the physical world, this could be equated with an attacker who targets a car with open windows and keys in the ignition rather than one that is locked and alarmed. One additional factor to consider in the attractiveness of a target is the perceived value of the exploit. In the car example, an attacker might pass up a 20-year-old, low-value car with the windows open, in favor of a locked and alarmed brand-new, high-value vehicle. This holds true in the application security space as well. Many enterprises create proprietary applications to fulfill their business needs. Sometimes these applications bring forward legacy data; other times, custom applications are developed to provide a business function that's not available in existing commercial off-the-shelf (COTS) product space. These proprietary solutions often have access to highly valuable enterprise data. If these systems are not secured—if they are the low-hanging fruit of the organization—the company's critical assets are exposed.

Going with an all-COTS solution isn't a guarantee of security. Many commercial vendors have had critical vulnerabilities in their applications. Some COTS applications are of particular interest to attackers because publishing a vulnerability about these applications often brings media attention and peer recognition. It can also lead to a "crack once, exploit multiple times" scenario. Some attacks, such as those which can crack passwords in Microsoft's NT LAN Manager (NTLM) challenge/response password authentication protocol or Cisco's Lightweight Extensible Authentication Protocol (LEAP), are so popular that attack tools are widely distributed on the Internet.

To say that application security is a new concern would be folly, because application security has been a concern since the first password-protected accounts were introduced. Application attacks have existed since the 1960's, and the MIT Laboratory for Computer Sciences "installed its first password-based system in 1977."5 However, the main security focus recently, beyond password and file controls, has been on network and perimeter security.

A proof point for this can be found in the body of knowledge related to what certified security professionals are expected to know about application security. One of the most well-known certifications for security professionals is the Certified Information Systems Security Professional (CISSP). CISSP certification is overseen by the International Information Systems Security Certification Consortium (ISC)2. The CISSP exam covers ten topic areas, and application security is only part of one of the ten topics covered. A complete list of the exam structure coverage can be found at CISSP.com. Since there are 250 questions spread out across the ten topics, for the "Application and Systems Security" domain one would expect no more than 12-13 questions cover application security specifically. And with all of the time enterprises spend trying to keep their distributed perimeters secure, the application security issue sometimes gets short shrift.

But as applications continue to malfunction, as attackers continue to seek the low-hanging fruit, and as applications manage and maintain more attractive information, attacks have increasingly focused on applications and the systems that run them. Many of the security stewards in enterprises have started to shift their attention to the source of many problems, the code itself, by trying to write more-robust, less-vulnerable applications.

If these applications are highly vulnerable to exploit or cannot otherwise be depended on to perform their intended functions, businesses can suffer financial loss. Arguably, a small application that has well-defined functionality and a finite number of inputs could be built to an extraordinarily high level of surety. But small applications that are well defined are not the norm for most code written today. Most applications are complex and comprise thousands or millions of lines of code, numerous components, and complicated environmental factors. Thoughtful design changes can often reduce the size of programs by an order of magnitude or more, and in some cases the resulting software is faster, more reliable, more functional, and easier to maintain. The security of these bulky applications is not only dependent on the code itself but also on the libraries the code calls, the components and systems the application interacts with, how users interact with the application, and the network environment in which the application runs. Due to this complexity, many enterprises accept that while code that is invulnerable to any and all attacks may not be a realistic requirement, the level of vulnerability can be managed and reduced.

Attempting to create code with no faults at all may not be possible, and at a minimum is very expensive. It also may not be desirable for some design methodologies because the time spent searching for flaws versus the consequences of those flaws may not be cost-justified. As Aldous Huxley wrote in *Eyeless in Gaza*, “Means determine ends; and must be like the ends proposed.” From a risk perspective, the intended “end” should relate to the level of surety required. High-surety applications have a high level of certainty that the intended properties of the application are met. Low and medium surety correspond to reduced certainty levels. Various surety levels are appropriate depending on the type and use of, and intended use for, applications. Very few applications require high surety, while many more require medium surety. It is not always necessary, or even possible, to find and eliminate all flaws in software code, before releasing applications.

It's Cheaper to Do It Right the First Time

Is there an inherent problem with fixing application software faults at the end of the development cycle or even in production? The Computer Emergency Readiness Team (CERT) Coordination Center at Carnegie Mellon University maintains a list of vulnerabilities, incidents, and fixes for systems and COTS applications. A quick check of the CERT database reveals that most major vendors (including Cisco, Microsoft, Symantec, Computer Associates, and Oracle) have had many vulnerabilities, caused by underlying faults, in their software. While it is impossible to find an equivalent source of information on proprietary, internal applications, there are security exposures that lead to patch issuance in many internal applications as well. The associated costs of exposures for proprietary applications must be gathered and assessed by the enterprises themselves.

Patch management costs impact the vendors who sell the COTS software because they must bear the hard costs of deploying a team to create a patch and then distribute the patch to customers. But vendor costs are minimal when compared to enterprise costs, which are multiplied by tens or hundreds of thousands of systems. There are also associated soft costs. When a vendor has a number of vulnerabilities in their software, consumers associate this vendor with unreliable or insecure code and may switch to another vendor. Although, the reality today is that rather than bear the expense of changing software vendors, most enterprises assume the cost of patch deployment and use layered techniques, such as configuration management and zoning, to prevent vulnerabilities for which there are no known patches. Consumers of COTS solutions bear costs associated with patching and reconfiguring faulty software and may also suffer loss through exploitation of the vulnerable COTS applications.

Finally, businesses that create their own code can experience financial impact from vulnerable code by having to deploy patch teams for their internal applications and possibly through exploit of the vulnerabilities. More information on vulnerability management and patching can be found in the *Security and Risk Management Strategies* report, “Vulnerability Management: Toward Technical Security Policy Management Products,” and the *Security and Risk Management Strategies* overview, “Enterprise Patch Management: Strategies, Tools, and Limitations.”

Developer teams are often under strict deadline requirements and anything that lengthens the release cycle can be seen as undesirable. Slowing down the process for a lengthy requirements and design session and then weaving security tests throughout the subsequent phases of the SDLC can be perceived as a release gate. Careful design and adequate testing, however, can actually accelerate the release cycle and also make it more predictable. Although a careful design phase may take longer, it can uncover potential problems earlier in the cycle when it is cheaper to fix defects. This reduces the financial risk of discovering serious problems, caused by defects, during unit testing. Using automated testing tools during the coding and unit testing process can reduce the need for costly and expensive fixes prior to or after release. But more importantly, perhaps as the step change associated with moving to the Internet ends, quality will become a critical driver for success in the software market.

Understanding Software Security

One of the problems enterprises face when developing secure software is a fundamental misunderstanding of how to create secure software. Robust software begins with well-defined requirements and thoughtful design. It is critical for designers and architects to understand not only what the application will be expected to do, but also the environment it will exist in, the other systems it will interact with, and the types of users who will access it. Moreover, the risk levels associated with the application and its functions need to be understood. More information on how to understand risk can be found in the *Security and Risk Management Strategies* overview, " Risk Aggregation: The Unintended Consequence."If designers and architects do not consider all of the above, how can they generate appropriate plans for developers to code to?

Placing an "application security" line item in the development plan is not enough. Designers must consider all aspects of how the application will be used and specifically detail for the developers what is expected. For example, one piece of highly sensitive information that is often used by application designers for identification purposes is the SSN. If SSNs are to be gathered by the application, is it necessary for them to be stored? If so, do they need to be stored encrypted? Can they be stored temporarily and then erased? Who (which users) and what (other applications or services) will have access to this information? If the application data is backed up for audit or recovery purposes, will the SSNs get backed up as well? If the information is shared with another system, will an audit trail to data map this sharing be recorded? Generating detailed use cases, data maps, and use models can reveal a number of unexpected paths in an application and catch potential vulnerabilities early in the process. More information on the ways that developers can be assisted in the enforcement of security rules can be found in the *Application Platform Strategies* overview, "Application Security Frameworks: Protecting Applications Consistently."

Being human, designers and architects will not be able to anticipate every potential problem with an application. Moreover, some problems may appear within code that cannot be anticipated by architects. Developers are rarely taught how to write secure code in university or on the job. The emphasis is on fast, efficient, elegant code, not on secure code. Another human problem is that our brains are especially adept at recognizing expected patterns but not so adept at recognizing unexpected ones. Application designers and developers may not be able to see faults or failures to provide input validation because they are expecting the code to work as they designed/coded it.

Another problem is that a lot of applications are an amalgam of components working together in a system. While a component on its own may be secure, when it interacts with another one, the end result may be an exposure. In their book, *Secure Coding: Principles and Practices*, Mark G. Graff and Kenneth R. van Wyk make this point using the UNIX "rlogin -l -froot" fault as an example. In this case, neither rlogin nor the application login were vulnerable individually. But when combined, the result was that an attacker could login in as root to an application that accepted rlogins, without the proper authentication. This problem has subsequently been fixed and is used for example purposes only.

The previous example is not presented as an argument against all code re-use. When done properly, code can be re-used securely. However, in practice, very often a component or code snippet is re-used without adequate consideration for the new environment and associated dependencies. This lack of consideration, as illustrated in the rlogin example, can lead to the introduction of security exposures.

Also, there are inherent problems with the programming languages themselves. C and C++ are commonly used languages in software development. However, they were not designed for writing large-scale applications; they were designed for writing operating systems and utility programs. C and C++ are compiled code, have access to memory, and are vulnerable to problems such as buffer overflows. Buffer overflows are the number one application vulnerability; they are caused by a failure to properly check input and can result in an attacker being able to execute commands on a system. Some of the most exploited and well-publicized attacks in recent years—Slammer, Code Red, and Blaster—were the result of buffer overflow vulnerabilities found in C/C++ application code.

Java is a semi-compiled language that compiles into byte codes that are then interpreted in a Java Runtime Environment (JRE), which can help prevent vulnerabilities like buffer overflows. Java uses managed code and provides library classes that are intended to help developers avoid common mistakes. Java developers can use a variety of extensions—such as the Java Cryptography Extension (JCE) and the Java Authentication and Authorization Service (JAAS)— that allow them to use already created and tested modules to supply security functions. However, Java developers are not exempt from thinking about security, and these libraries are neither perfectly written nor applicable to all uses. For example, if classes and methods are not explicitly defined as private, they could become an attack vector.

At most enterprises, code goes through a software quality assurance (SQA) testing phase. SQA testing generally involves setting up controlled environments and running software through tests designed to confirm that the software performs as expected. For a simple calculator application, SQA would test that the numbers entered by the user are accurately captured by the application and that the mathematic functions (such as addition and subtraction) are performed accurately in some test cases. What is not always captured in the SQA testing, however, is unexpected misuse of the system or all combinations of parameters. Misuse does not mean the acts were intentionally malicious, although they could well have been. Rather, it is about how a system behaves when used in a way the designers and developers didn't intend. While careful design and coding should reduce misuse errors in the final code, it is important that code be tested for errors prior to releasing for general production use.

What About Managed Code?

While managed code is not a new concept, it has gained popularity in recent years with the introduction of Java and .NET. The high-level intent is to create an environment that will help developers write more robust code and to protect users, at run time, by running the code in a controlled environment. Many routines that developers traditionally have had to write anew for each application (e.g., garbage collection, bounds checking, and exception handling) are managed by the controlled runtime environment. The theory is that by automating these routines developers will be able to write code more quickly and more securely.

The conceptual benefits of managed code and protected runtime environments are attractive. The code can be ported to any platform that supports the runtime environments. And freeing developers from common routines, such as those mentioned above, will most likely streamline development. However, it is neither a silver bullet for creating secure applications nor a replacement for sound design and testing. More information on how managed code relates to the creation of secure applications is covered in “The Details” section of this report. More information about managed code in general can be found in the *Application Platform Strategies* Reference Architecture Technical Position, “Application Platform Foundations.”

Code-Scanning Tools

A number of automated code-scanning tools can help enterprises produce more secure code. It's important to understand the terms “software fault” and “software failure” in software development parlance. A software fault is a problem, or mistake, in the semantics of the code. Though a software fault is a problem, it will not necessarily cause software failure. Software failure is when the program does not run as expected. A fault, or a number of faults, can cause failure. Faults and failures do not guarantee that there is a security vulnerability in the code; however, they often point to potential problems in the code that could lead to vulnerability exposure. In addition to this, software can be vulnerable to attack even if there are no underlying faults in the code. The design of the application itself could be flawed.

Code-scanning tools can be divided into two broad categories: static analysis and dynamic/runtime. Each type of tool provides a different view of the application's security state. Static tools find faults via review of the source or binary code itself, looking for known programming errors such as lack of input validation checks, formatting analysis and data handling. Static analysis employs formal methods, mathematical techniques to identify potential problems, and other less rigorous techniques that codify rules of thumb. Model checking in static analysis confirms if a piece of software conforms to a given set of requirements. Abstract analysis in static code checking uses semantics to describe and test how an application will execute. Dynamic runtime tools discover failures, monitor the application when it is running, and help with understanding contextual issues such as how the program interacts with other systems and services, if there are memory leaks, and how the application impacts the performance of the system on which it is running. In theory, static checking can be perfect, but in reality, it can be very expensive and difficult to do well.

One of the reasons many enterprises opt to use both types of tools is that they can be used at different points in the development process. Source-code analyzers can scan source-code components during early stages of development. Runtime tools, by definition, require that applications be running and therefore, in most cases, further along in the development process.

In the static-analysis space, the two broad categories of tools are source-code and binary scanners. While source-code scanning may have more insight into specifics of the code itself, such as comments and variable names, it requires having the source code available. Binary scanners can be used on any binary, which in some cases may be all that a company has available to it. This is particularly true for companies purchasing COTS software that the vendor may not be willing to release the source code for. It also applies to many patches for which, again, the user organization does not have access to the source code. In addition, in some organizations, business units, subsidiaries and subcontractors may not be willing to share the source code, so binaries may be the only option for scanning. Be aware that if binary code scanning disassembles or reverse-engineers programs during analysis, their use may violate software license contracts or federal law.

In addition, there are at least four broad users of these tools: the developers, auditors, attackers, and researchers. Developers are tasked with getting the code written; they are often crunched for time and require a tool that can run quickly in their development environment. Auditors are concerned with testing and approving the application at an appropriate security level. Auditors most often work outside the development environment and, based on their function, may be allowed more time to run scanning tools. Attackers may use these tools to find faults in programs and to verify that their attacks will not be detected in use and to gain insight into systems and operations. Researchers use these tools for understanding issues, testing theories, and developing new knowledge.

Why Code Scanning Is Not Enough

Although code-scanning tools are an important part of a secure SDLC, they are not a replacement for other steps. Again, it takes a village to write secure applications and requires input from a variety of stakeholders and active participants in the process. First and foremost, appropriate application security needs to be part of an organization's critical criteria for success. This does not mean that all applications must be at the highest level of surety. It does mean that appropriate team members should understand the unique requirements and build these requirements into the application's design and architecture. If low surety is acceptable for an application, how does that requirement translate into the design? And how is the low-surety (or medium- or high-surety for that matter) requirement documented so that if the application does fail and cause loss to the enterprise, appropriate accountability can be placed? Surety levels also have to be associated with components like re-usable code and libraries if their use is to be matched to the need, and this introduces life cycle documentation and change control requirements.

If management accepts the need for security in the SDLC, a number of additional steps can be taken to make this a reality. For example, does the enterprise want to invest in designer and developer training? As noted above, few universities are teaching secure coding practices. There are, however, classes that designers and developers can take to help them understand security concepts and how to model and create more secure applications. This requires an investment in time and money from the enterprise and also on the part of the application developers themselves. Will the development team be rewarded for their time? Will they be rewarded for writing more robust code? If not, the enterprise will continue to release code with a high number of faults. Coders deliver to requirements, and employees deliver what management rewards.

The Pareto Principle and Application Security

At the beginning of the twentieth century, Italian economist Vilfredo Pareto noticed that 80% of the wealth in Italy was owned by 20% of the population. Joseph Juran, an early visionary for quality management, applied the "80/20 rule" to quality management, positing that 20% of the quality variables result in 80% of the quality problems. Although an exact 80/20 may not be consistently provable in software security, it does illustrate one of the compelling arguments for using code-scanning tools. A large portion of software failures in applications written in C and C++ are caused by a small number of common coding errors (particularly buffer overflows caused by lack of input validation). Many of these errors can be caught by implementing proper design practice, weaving security throughout the SDLC, and judicious use of code-scanning tools.

Code scanners themselves have limitations. In 1936, British cryptographer Alan Turing proved the "halting problem." In computer science, this means that it is impossible to know, for certain, if an arbitrary application operating in a Turing machine will complete (or halt) for all possible inputs. Taken to a more applied level, this means that it is theoretically impossible to validly test results for every possible input sequence in certain classes of systems. In other words, except in select cases, code scanners can't provide 100% assurance that the application is without vulnerabilities or even has been coded properly to the design specifications. The fact that code scanners alone can never detect all software problems is not a reason to abandon them. It is, however, a reminder that complete trust in code scanners can create a false sense of security and may put applications and the enterprises they support at risk. Code scanners are part of the application security process; they are not the entire application security process.

Market Impact

Code-scanning tools have been in use for a number of years, but it is only in the past five or so that the market has blossomed for specific security-aware code-scanning tools. A number of factors have contributed to this market trend. One of the most important reasons organizations are concerned with secure code is that they are increasingly dependent on software. Many new-model cars are shipping with up to one million lines of C code in them. Everything from coffee makers to microwave ovens to phones have embedded code running inside of them. And businesses themselves are dependent on software to manage everything from the supply chain to the sales force to the financial reporting.

Market Drivers

In addition to software dependency, fears caused by security failures (software and otherwise) are contributing factors. Security professionals aside, most humans do not live in a constant state of hyper-awareness, waiting and watching and planning for what can go wrong. Modern humans in Western societies tend to respond to risk only after a failure or attack has occurred. An illustration of this is home security systems. The majority of these systems are installed *after* a robbery or attempted robbery has occurred. After 9/11, although the attacks were not software failure-specific, many people who had not previously concentrated on security awareness became very security-aware. The highly publicized worms of 2001, CodeRed and Nimda, and in 2003, SQL Slammer and Blaster, only served to fuel the concern. These worms illustrated, in a real-world manner, the costly business impacts that can occur when non-secure code is exploited.

In the cyber world, this translated to many enterprises raising their security awareness and trying to put more attack-resistant infrastructures in place. On January 15, 2002, four months after the September 11 attacks, Microsoft Chairman Bill Gates released what is now commonly referred to as the “Trustworthy Computing” memo. In this memo, Gates referenced September terrorist attacks and stated that achieving the goal of trustworthy computing was the highest priority for Microsoft in 2002. Gates called for changes in the way Microsoft developed applications to ensure that the programs could deliver accountability, security, and privacy. Microsoft is a strong voice in the software industry. When the company pledged to provide better application development, many enterprises took note, not simply in anticipation of more robust MS applications, but also internally to address their own development lifecycles.

In 2003, the Trusted Computing Group (TCG), a consortium of vendors and both for- and non-profit organizations, was created to help promote and develop hardware and software interface specification standards to help designers implement higher-surety systems to protect their customers. While the TCG does not provide provisions for code surety, it does provide verification that code is unaltered and meets vendor specifications.

Government regulations have been another driver toward the growth of the secure code-scanning market. Although there is no prescriptive requirement in regulations such as Sarbanes-Oxley or the Health Insurance Portability and Accountability Act of 1996 (HIPAA) for integrating security into the SDLC, the regulations require that controls be placed on information to ensure that it is valid (and in the case of HIPAA and SB1386, that it is protected). Because data, financial information, and healthcare information are often stored in digital form in one or a number of applications, there is an implicit requirement to ensure that these applications are reliable and will perform their duties as expected. Moreover, these applications need to be resistant to attacks to help ensure that malicious users do not alter the integrity or privacy of the data.

Another major hot-button issue for application security is outsourcing. The same approaches to creating secure applications should be used regardless of whether a developer is a full-time employee working in corporate headquarters or an outsourced resource in another location or country. However, in practice, the move toward outsourcing has created serious concerns regarding application development. One of the reasons for this is that when the development is outsourced, it can be harder to enforce some of the same measures on code creation. Often a requirement is shipped to the developer who then codes it up and sends it back. Upon receipt of the code, it can be scanned for potential errors before being integrated into the rest of the application.

The industry is becoming increasingly aware that code not written in a robust manner can have far-reaching impact when it fails. This is especially true as companies continue to expand their reach to external users, customers, and partners through the Internet and web services. This breaking down of boundaries and concern for stronger application security has increased demand for tools that help enterprises create more secure code and do so more cost effectively. This demand has created the opportunity for the code-scanning tool market.

Where Is the Market Heading?

Most of the available tools are offered by vendors who are focused on one thing: supplying tools to help scan and test code. Security code scanning is a fairly immature market. Most of the leaders in the space (e.g., Fortify, Klocwork, Ounce Labs, and Secure Software) are less than five years old. As with most innovative technology, advances are being made most quickly in the start-up space.

As the technology and market matures, Burton Group expects that large, established vendors who supply complementary technologies will either develop their own tools or add one of the startups to their portfolio. At the June 2005 Tech Ed Conference, Microsoft announced that the next version of Visual Studio, (currently slated for release sometime in late 2005) will include integrated, security-aware code-scanning functions. This technology was not created by Microsoft; it was built by SPI Dynamics, a web application security scanning company. IBM is already partnered with Secure Software, and acquiring that company or another static-code analysis vendor would complement IBM's Rational software development platform offering. Computer Associates' AllFusion line of development tools is another good fit for security-aware code-scanning functionality. While it is logical to expect major integrated development environment (IDE) vendors with development portfolios to begin offering these tools as part of their IDEs in the next 3-5 years, enterprises should not wait until that time to begin using the tools that are now available from scanning vendors.

Market Segmentation

There are two distinct types of scanning tool vendors: the ones that concentrate on the C, C++, and Java market, and those that focus on scanning and securing web applications. For the most part, a vendor that supplies one type of scanning tool does not have an offering for the other. Additionally, the buyers of these tools tend to be members of distinct groups. For this reason, this report concentrates specifically on the C, C++, and Java tools rather than covering the web application market.

The current C, C++, and Java scanning market is divided into two broad classes: static scanners and runtime analysis. Source-code scanning tools are available as stand-alone products and offered as plug-ins to IDEs. Static analysis tools are also available for binary scanning.

Typically, an IDE offers code creation and editing, a compiler, an interpreter, and a debugger. Languages and features supported in an IDE vary depending on the provider. IDEs are offered commercially and as freeware. An example of a freeware IDE is Xcode, which is available for the Macintosh OS X platform. Commercial IDEs include Microsoft's Visual Studio suite which supports C, C#, and Visual Basic. Sun has an IDE for Java called Sun ONE Studio. Borland has IDEs for both C++ and Java, known as C++Builder and JBuilder, respectively. Eclipse is an extensible OpenSource IDE for a number of languages, including C, C++, and Java. Some IDEs have code-scanning checks built in, but no IDE yet constructed can perform in-depth security code scanning.

However, third-party tools can be run inside the IDEs to perform security scanning checks. In general, developers use scanners inside the IDE and audit/security teams use stand-alones. Developers often have a speed requirement associated with delivering completed code, so tools that work fast are desirable. Scanners in the IDE are often optimized to run quickly and deliver low overhead to developers. Audit teams are usually driven more by the requirement to ensure the soundness of the code than to deliver it quickly. Auditors use stand-alones because even though they often take longer to run, they provide deeper analysis of the code.

Static code-scanning vendors that entered the market early concentrated on quality management; these vendors include Coverity, GrammaTech, Klocwork, and Parasoft. A slightly newer class of code-scanning tools vendors, including Fortify, LogicLibrary, Ounce Labs, and Secure Software, entered the market with tools specifically aimed at scanning code for security problems.

In addition to the tools vendors, a handful of independent companies, such as Cigital and Security Innovation, provide code scanning and SDLC process and security improvement guidance as consulting services.

Market Dynamics

The dynamics of the secure code-scanning market are evolving in parallel with the demands on the market. Although code-scanning tools cannot trap every error, and may produce large numbers of false alerts if not properly tuned, many faults in applications can be traced back to a small number of design and programming flaws. Code-scanning tools can detect many of these flaws in an automated manner, which makes their use justifiable for companies that want to deliver higher-surety applications. Development models have also evolved considerably in the past few years, and the code-scanning tools for security have been, by and large, created to meet the changes.

Recommendations

The way an application is developed, from initial requirements-gathering through final pre-production testing, can greatly impact the cost and security of the resultant application. Catching problems earlier in the process is cheaper than catching them later, especially during post-production. Therefore, Burton Group recommends weaving security throughout the entire SDLC, from design through production, remembering that the means determine the ends.

Begin the process with a thoughtful requirements-gathering phase in which security concerns and acceptable levels of risk are considered. (More information on how to determine appropriate surety levels for applications can be found in the *Security and Risk Management Strategies* report, "Building Secure Applications: How secure do you want to be today?.") Carry this approach throughout the SDLC. Spend time on requirements, architecture, and design. Application development is usually an iterative process. Using a methodology, such as spiral—that allows for a revisiting of earlier phases, and re-design or architecture if required—provides more flexibility and can address security concerns at the root source rather than as a stop-gap measure implemented after the fact to make up for a flawed design.

Process and Accountability

What does a good SDLC process look like? A number of methodologies are available to enterprises. If an enterprise does not currently have a documented SDLC, it should create one or adopt an existing model. Additional discussion of models, such as the waterfall and the spiral, are discussed later in this report. At the highest level, the software development process can be broken down into the following components:

- Requirements
- Risk and cost-benefit analysis
- Architecture and design
- Implementation
- Testing and acceptance

In order for a product to be at the correct surety level, security and quality considerations need to be included in every phase of the SDLC. Those quality and security considerations must be oriented toward assuring the protective efficacy of the implementation. Someone must be accountable for checking and signing off on security during each phase. A point that is often lost is that testing a product for functionality is not the same as testing it for security exposures. Most test cases are designed to confirm that the product is operating as expected. But security checks must go beyond functionality to test for potential misuse and abuse.

Requirements

The requirements-gathering phase is concerned with answering the "what" questions, such as "What problem is being solved and under what conditions?" During the requirements-gathering phase, the security of the application should be considered and defined clearly. It is especially useful to consider who or what will be using the application. Will it be humans, other applications, or both? Modeling use cases will help to contextualize the application as it will run in its production environment and may also help to expose, early in the process, where more or less security is needed and if levels of security cannot be met. To some, less security may sound like a bad idea, but more security is not always better. Let's take a look at a simple example: password length and aging. It's arguable that a password that is aged every 24 hours, that must be 40-characters long, yet cannot be composed of any strings longer than three that comprise a word found in the dictionary, is a strong password. However, if we expect humans to supply this kind of complex password to an application, the ease of use will be greatly affected and the security of the password itself could be at risk. Why? Because humans aren't great at remembering long strings of characters. If the string changes every 24 hours, most people will resort to writing the password down somewhere. A password written on a sticky note or stored in drawer is less secret than one that is stored exclusively in the user's brain.

The ways in which the application interacts with other applications and the operating system (OS) on which it will run also need to be understood during the requirements-gathering phase. If the application will share data with another application, how will the applications authenticate to each other? If mutual application authentication is required, it needs to be defined. The way authentication is executed in the final product can be defined during the design phase, but such a requirement needs to be documented early on. If the application accepts data inputs from another application, does that input need to be validated? These are just a few examples. Clearly, during the actual requirements-gathering phase, there will be many questions that need to be answered. The key is to think through the application's uses, its users, its performance, management, and storage requirements, the locations of its input and output, and ways to document all of the requirements so the designers and architects can meet them. More information on frameworks and approaches a developer should use to implement security protections in an application can be found in the *Application Platform Strategies* overview, "Application Security Frameworks: Protecting Applications Consistently."

Risk and Cost-Benefit Analysis

With the documented requirements in hand, a risk- and cost-benefit analysis should be applied. Sometimes a requirement may not be cost-effective for an enterprise. An example is a requirement for mutual application authentication using x.509 certificates. Implementing a public key infrastructure (PKI) in an enterprise is a costly endeavor, and if the cost of using the certificates does not match the risk reduction they provide, the solution is inadequate to the need. It's important to understand that there are no single right or wrong answers and that one size definitely does not fit all. There are better- and worse-fit answers for a given enterprise and application. It is the job of the people doing risk management to understand how the company works, what the risks and priorities are, and which requirements are essential for the end product. If, during this phase, it is determined that some of the requirements cannot be met, those exceptions must be documented and the previous phase iterated appropriately.

Architecture and Design

The next phase in the SDLC is to generate the architecture of the application. Again, security considerations are a critical part of this process. The application should be characterized at an architectural level and internal and external interactions clearly understood and documented. The architecture phase is critical because it is where the requirements are translated into documents that the developers will work from. If the architecture fails to properly structure the design for effective protection as specified by the requirement needs, then the resulting application will never be able to fully meet those needs. The architecture phase is when architects must think through the means by which technical security requirements will be accomplished. Using inputs as an example, requirements must state which classes of inputs need to be validated to certain surety requirements. The architecture must define how those inputs will be validated to the same surety requirements. Project managers and application architects need to be aware that handing vague design documents to the developers runs a high risk of resulting in poor implementation. It is not reasonable to expect application developers to invent security where none has been defined.

Implementation

The implementation phase, or coding, is the next step. Here, even with clear, well-thought-out designs, there are still potential areas where application insecurity can be introduced. One way to help prevent the introduction of vulnerabilities through coding errors is to have a well-trained development team that is aware of common problems in C, C++, and Java code. Carefully checking all inputs to the system is a good start. Avoiding common errors—such as using the strcpy or strcat functions that do not check input lengths—will help as well. During this phase, using source-code scanning tools to catch errors that developers were not aware of will help strengthen the end product. Some enterprises may also wish to utilize manual source code review using internal or external resources. Code scanners are limited to catching flaws they know about. Sometimes having an experienced, security-aware developer eyeball the code uncovers errors the scanners can't.

One potential “gotcha” during the coding phase is the re-use of code and the use of libraries. Code re-use, often simple cut-and-paste, is a common practice in the development world. While it makes sense from an efficiency perspective to use existing code rather than create it anew each time, it is not without potential for security problems. If the design of an application was not completed with security requirements in mind or the code in question has faults, the application it is likely to have flaws or failures. Also, some code is not re-usable in certain circumstances. If code is optimized to work with certain hardware and re-used on other hardware, problems can result. Libraries provide a more codified approach to code re-use. They are classes of specified features and functions that a developer can call in the application. The intent is that the libraries themselves are stable and will not introduce flaws into the application. But if the libraries are not stable, calling them in an otherwise robust application brings in the libraries' faults.

Testing and Acceptance

Security testing, even with a careful design and scanning of the source code, is not over when the source code is finished. Testing the compiled code running in an environment that emulates the final production environment can expose additional, unexpected behaviors. During all security testing of the application (both in source and runtime), it is important to map identified problems back to the initial requirements and design. Otherwise, the problems are uncovered in a vacuum, without context. Runtime analysis tools monitor the application as it executes and can catch issues such as memory leaks that often relate to insecurities in the underlying code. Other tools that can help in testing the strength of the application include application penetration testing tools, fault-tolerance injection, and vulnerability scanning. Again, these tests can be run by internal or external groups. For the vulnerabilities that are deemed unacceptable in the final product, re-coding is in order. This means iterating the development cycle and performing regression testing on the new source and compiled code. The enterprise may choose to accept vulnerabilities that are discovered during this phase. Documenting these vulnerabilities and the reasons for accepting them builds accountability into the process. There may be potential liability associated with finding—yet failing to fix—known security faults. This should be considered in the risk acceptance process.

When the application has passed the testing and acceptance phases, it is ready to be deployed. But this does not mean the security process is complete. Applications in use can behave in ways they did not during testing. Some applications behave differently when processing large amounts of data, when there is a heavy load on the application, or due to unexpected environmental issues. If the design and implementation team made the assumption that the application would always run behind a firewall that is blocking certain ports, and then it is placed in an environment that does not have such a firewall, the application could fail in an unexpected manner. The application needs to be monitored in production to ensure that it is performing properly.

Another fallout from launching a product into production use is the discovery of vulnerabilities that require a patch to the deployed product. Creation and deployment of patches must be undertaken with the same security-aware SDLC process that the original application was. More information on patch management in the enterprise can be found in the *Security and Risk Management Strategies* overview, “Enterprise Patch Management: Strategies, Tools, and Limitations,” and the *Security and Risk Management Strategies* report, “Leading Patch Management Vendors Prepare for Change As Products Converge.”

The above-outlined approach to patching is not meant to suggest that if the SDLC itself is seen as flawed, it should not be tweaked and optimized. Rather, it suggests that patches are code and need to be developed to the same standards that the application was. Very often, there is a tendency to rush the patch in order to plug the vulnerability as quickly as possible. But this can lead to introducing more faults. Having a process in place—with checks and balances that tie back to accountable parties and sign-offs—may sound more arduous than simply getting the code written and into production as quickly as possible. But careful testing of the patch can help reduce problems after the patch is issued.

Test Early and Often

Testing is not something to be done only at the end of the product lifecycle. For most applications, this approach is too little too late. To achieve optimal application reliability, begin the testing and audit process with the requirements phase and continue to test throughout the SDLC.

In order for this approach to work, an enterprise must understand that accountability for the testing needs to be assigned. And because humans are flawed creatures, it is unreasonable for developers to be 100% responsible for the security of their code. Consider assigning a specific team for application security auditing; for example, a group that is trained on secure coding and application development techniques and has no responsibility other than to ensure that the final application meets the enterprises' security requirements.

Use static analysis tools judiciously throughout the SDLC. Provide tools, such as scanners that run within the IDE, to help developers catch problems during the development phase. Use of managed code, if appropriate for an organization, should be viewed as another method of code scanning in the IDE. Managed code does not obviate the need for secure design, but it can help alleviate some of the developer overhead when creating code. Also, provide auditors with tools, such as static analysis scanners and runtime analysis that can be used to confirm that the applications are meeting requirements. While no single tool (or even set of tools) will catch all problems, tools are a valuable addition to the audit process.

Pick a Tool

The decision about which code scanners to use will ultimately rest on an enterprise's review of the code-scanning options and comfort level with the various tools. When picking a tool (or tools), the enterprise should determine the users' business and technical requirements and the views those users require. Also, understand what the company wants to accomplish with the tool. Does it need to work within an existing IDE? Does it have to be a portable stand-alone for mobile audit teams?

While the security specific code-scanning vendors share many similarities in approach, they do have different areas of focus which may impact an enterprises purchasing decision. For example, Ounce Labs provides an IDE scanning tool but is more heavily focused on providing tools for auditors. At Secure Software, the tools are augmented by an entire lifecycle approach called Comprehensive, Lightweight Application Security Process (CLASP).

Another differentiator is source versus binary scanners. Some enterprises may not wish to scan binary code, while others, particularly those that are reviewing COTS software, may have a need for binary code scanning. Companies that already have non-security specific code scanners from vendors such as Klocwork and Parasoft may find that their security needs can be met sufficiently with one of these tools.

The Details

This section details why fixing defects earlier in the software development lifecycle (SDLC) is cheaper, weighs the merits of common programming models and testing, and reviews common implementation patterns associated with code-scanning tools. A taxonomy of market leaders in the C/C++/Java application security space concludes the section.

Is It Really Cheaper to Do It Right the First Time?

Studies have shown that it is cheaper to find and fix defects earlier in the SDLC. In the report, “The Economic Impacts of Inadequate Infrastructure for Software Testing,” published by the National Institute of Standards and Technology, two studies of relative cost to repair defects are compared (see Figure 1).

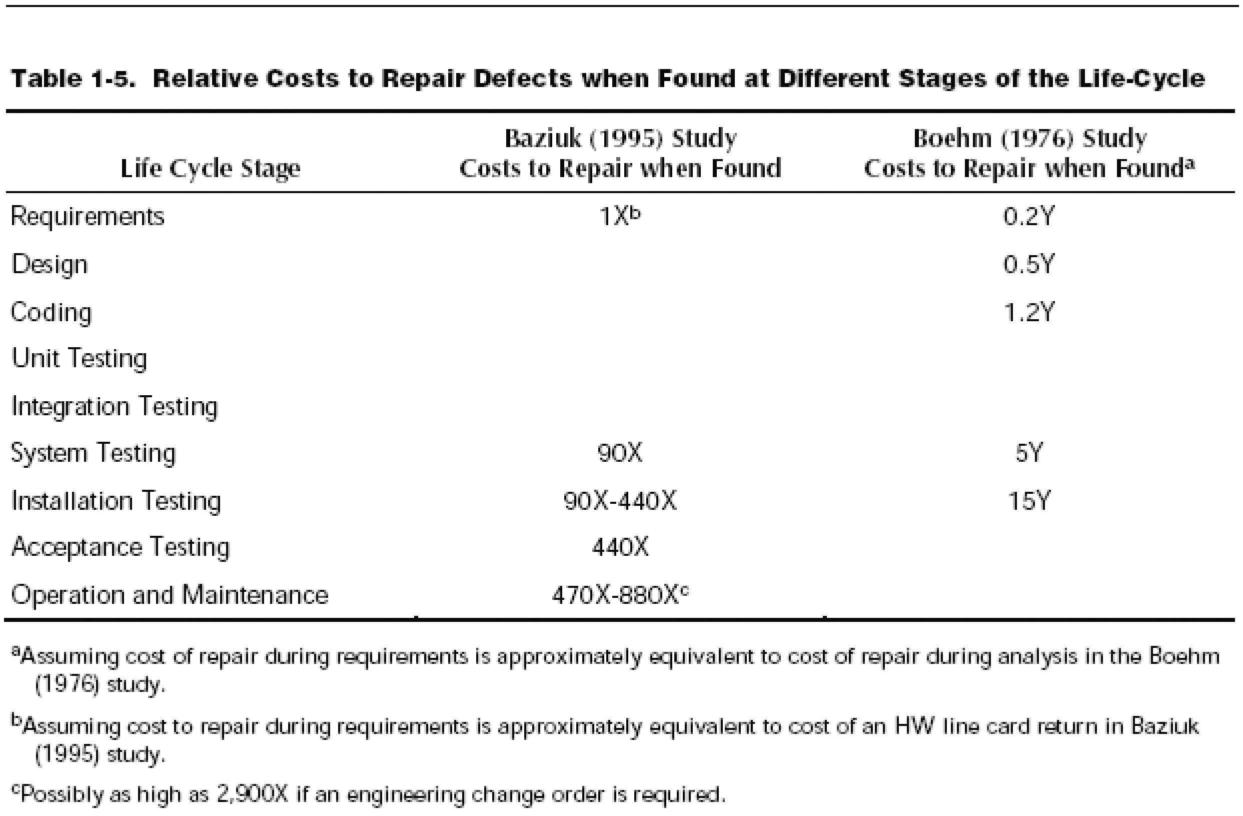

Table 1-5. Relative Costs to Repair Defects when Found at Different Stages of the Life-Cycle

Life Cycle Stage	Baziuk (1995) Study Costs to Repair when Found	Boehm (1976) Study Costs to Repair when Found[a]
Requirements	1X[b]	0.2Y
Design		0.5Y
Coding		1.2Y
Unit Testing		
Integration Testing		
System Testing	90X	5Y
Installation Testing	90X-440X	15Y
Acceptance Testing	440X	
Operation and Maintenance	470X-880X[c]	

[a]Assuming cost of repair during requirements is approximately equivalent to cost of repair during analysis in the Boehm (1976) study.

[b]Assuming cost to repair during requirements is approximately equivalent to cost of an HW line card return in Baziuk (1995) study.

[c]Possibly as high as 2,900X if an engineering change order is required.

Figure 1: *Relative Defect Repair Costs (Source: National Institute of Standards and Technology)*

Although the two testers found differences in actual relative costs, both studies show that the cost to repair defects increases dramatically the later in the cycle they are undertaken. At the National Science Foundation (NSF)'s Center for Empirically Based Software Engineering (CeBase) METRICS02 Workshop, researchers agreed that:

- Finding and fixing a *severe* software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase
- Finding and fixing *non-severe* software defects after delivery is about twice as expensive as finding these defects pre-delivery

- Finding and fixing software defects after delivery is significantly more expensive when the maintenance work is being done by an organization other than the one that developed the system6

While the exact costs associated with fixing defects differ, it is clear that the cheapest time to repair a defect is during the requirements and design phases.

Research shows that fixing a defect early is cost effective. But will fixing defects reduce vulnerabilities? A vulnerability in an application is not the same as a defect or a fault. However, it is commonly accepted that faults in application code lead to vulnerabilities in the resultant end product. Although it has not been proven in referenceable research that fixing defects leads to a direct reduction in vulnerabilities, it can be extrapolated that fixing defects early leads to a more cost-effective development model and less-vulnerable applications later.

Development Models

How an application is developed can directly impact the ease with which security is woven into the SDLC. To cover all the choices that must be considered when selecting an appropriate SDLC methodology would be too complex an endeavor in this report. However, this report does note some of the most commonly used methodologies and indicate how the chosen model can impact an application's surety level.

The most traditional development model is the waterfall model, a linear approach that works in sequence. Once a phase in the SDLC has been completed, it is closed and cannot be revisited, like water flowing off a cliff. Looking at this model from a security perspective, some potential problems can be seen. If unit testing uncovers a design flaw, but the design phase is completed, the application developers may be required to fix the flaw within the code or—using an environmental factor, such as a wrapper—in production. Based on a cost-benefit analysis, this may be reasonable for the enterprise. But an SDLC model that is more flexible will allow root causes for problematic software to be addressed in design by iterating the process.

The rigidity of the traditional waterfall model has led to the development of a number of evolutionary models for software development. The modified waterfall model looks very similar to the pure waterfall model but removes some of the sequential limitations by overlapping some of the phases. If a problem with the immediately preceding phase is found during the initial launch of the subsequent phases, changes can be made before advancing. If problems are found once a phase has been closed, there is still a restriction on re-opening previous phases.

The spiral methodology is an iterative method introduced by Boehm in 1986 to address the need for a risk-driven approach in the SDLC.7 There are distinct phases, as in the waterfall model, but they can be iterated, or revisited, throughout the SDLC. This approach makes the spiral model more agile than a rigid waterfall. If a later phase, such as runtime testing, uncovers a flaw in the design architecture, the development team can assess the severity of the flaw and, if needed, revisit or retool the design.

One of the most recent methodologies to emerge is the Extreme Programming (XP) model. Some might argue that it could be called “Extreme Spiral” because it is an iterative method, but the iterations in XP are much faster than they are in the spiral model. In XP, programmers work in pairs, and code cycles and revisions are very short. XP promotes code prototyping and testing rather than a long, involved requirements and design process. Components are built and integrated frequently (sometimes many times in one day), and the design of the final application is revised and tuned throughout the entire development lifecycle. Due to the rapid changes in design, documentation is usually not completed until the application is done. Since a number of security issues can be revealed by examining early design documents, the lack of documentation can lead to security exposures in the final code that could have been caught during design. And while XP does incorporate the concept of use cases, it does not account for “abuse cases”—instances in which the application fails or is attacked. As discussed previously, use cases can help audit teams determine early in the process if there are paths or dependencies that might lead to application failure (abuse cases).

One very cautionary note about XP: The term is often misused to cover a lack of SDLC methodology. Ad hoc development (i.e., programmers writing code without understanding requirements) is not extreme; it is chaotic. However, if XP is approached methodically, and the programming pairs have been properly trained in secure coding practices, and the unit tests and threat models have been carefully constructed, XP could be a viable option for some development shops.

An interesting, emerging development model is the highly modularized Aspect-Oriented Programming (AOP). AOP has not reached mainstream adoption yet. But for readers who would like to investigate this methodology in more detail, further information can be found in the *Application Platform Strategies* overview, "Ghosts in the Machine: Aspect-Oriented Programming."

How Static Analysis Works

The earliest static code analysis tools did little more than match byte code to known programming defects or flaws. One of the earliest, ITS4, a freeware tool from Cigital, was a glorified grep tool that did little more than look for a certain type of string in the code and then return an alert flag. The term "grep" comes from the UNIX grep utility that searches for patterns in strings and files. Because these tools were of limited use and returned a number of false positives, the market has evolved to a more contextualized checking approach. Most of the large vendors still supply grep capabilities, but these are augmented with contextualized analysis. In context, potential flaws can be better understood and false positives reduced. The evolved approach more closely resembles the work that compilers perform. Context-based analysis goes beyond simply looking for certain strings to understand where arguments were passed from, the data flow, how data is passed from class to class, and so on. This context-based approach enables the scanners to understand the vulnerabilities in a more comprehensive manner.

In addition to contextualizing the information returned, many of the available tools differentiate themselves based on the intelligence of the information returned. Rather than simply flagging a potential problem, the tools attempt to deliver easily understandable results that can be acted on. The goal is to make the output from the products more useful for developers and auditors. For most developers and auditors, a simple flag that there is a potential problem is less useful than a contextualized analysis of why the problem exists, what impact that problem could have on the final application, and how to approach correcting the problem.

An important consideration is what the scanning developers, and therefore the scanning tools, "know." These products are programmed to look for mistakes that have been seen in past applications; they are not capable of finding mistakes they are not aware of. In Java code, this can include looking for problems such as null pointers or an ignored return value. In C and C++, the analyzers look for commands that often lead to application failure, such as the use of *strcpy* and failure to properly validate input data.

Code-Scanning Evolution

It is important for enterprises to remember that in order for code scanners to be an economically sound purchasing decision, they must provide value that is greater or at least equal to their cost of acquisition and ownership. This fact has placed a burden on the vendors to provide tools that can show some tangible utility. If the application security problem could be solved with other tools, such as wrappers and gateways, then there would be no need for scanners.

Some of the earliest code scanners were glorified grep; the outputs were voluminous and hard to understand. Because context was often ignored, the output resulted in a large number of false positives—potential errors that were determined not to be actual problems when the complete context was taken into account.

Most next-generation scanners are well past the grep-only phase, and concentrate on providing readable and actionable information. But they are still not a replacement for security-aware design, understanding from a business perspective what level of surety is required for the final application, or how the application will be used in its production environment.

Black, White, or Gray?

When using tools to scan and test code two high-level approaches are used: black box and white box. Before going into a discussion of what these terms are generally accepted to mean, it should be noted that the terms apply to the underlying testing methodology rather than to the specific tool used. The approaches relate to how much information about an application is known prior to running the tests.

The black box testing approach is akin to a blind penetration attack on a network or system. The application is viewed as a black box—nothing about the internals of the application is known to the tester. Black box testing emulates what an outsider would see of the application because only the interfaces are available. The benefits of this approach are that it gives the tester an outsider's view of the application and it can be performed on applications for which the source code and internal operating parameters and state are not available. In some cases, often when buying COTS software, the acquiring enterprise has no access to the source code but may still wish to perform some form of application vulnerability analysis. Binary code scanners can be used in black box testing as can application vulnerability scanners. In this situation, more information is available than in a strict black box test because the executable code and program state are retrievable.

White box testing methodology understands all of the system internals, how the system is expected to work, and what the requirements and underlying design are. Very often, the source code itself is the subject of white box testing, but not always.

Gray box testing is often used internally—some knowledge of the system internals and expected requirements may be used during different phases of the testing process. There is no perfect way to test an application, but using a blend of the various methodologies shows different views of the application at different stages. In general, white box, informed, testing can be used throughout the SDLC. Black box testing is used when the application is complete but before it is launched into the final production environment.

Managed Code and Virtual Machines

The increased focus on creating more secure applications has many enterprises looking at developing code in managed code environments such as the Sun Java Studio, NetBeans, and Microsoft Visual Studio for .NET. This begs a few questions: Is implementing a runtime environment (such as the Java Runtime Environment [JRE] or the .NET Common Language Runtime [CLR]) inherently or appreciably more secure than running an application directly in an operating system? And will developers really be able to develop more secure, less flawed, code in managed code environments?

Java code is highly portable; as long as a JRE is supported by an operating system, the code will run. Currently, Sun supplies JREs for Windows, Solaris, and Linux. Apple provides a JRE for Mac OS.

Microsoft has also introduced managed code within its .NET framework. Using .NET, application developers can create applications in C#, J#, C++, or Microsoft Visual Basic. Regardless of the language chosen, all .NET developers can make use of a common set of class libraries. At runtime, the code is executed in the CLR on Windows platforms. At present, support for the CLR on non-Windows platforms is available from the Mono Project, which is led by Novell. Mono supports .NET for Netware, Mac OS X, BSD, Solaris, Linux, and Windows platforms. But it is not clear if other CLRs will be introduced for non-MS Windows platforms or if use of the CLR will ever gain significant traction outside Windows platforms.

Although the goal of managed code is a laudable one, in practice, using managed code does not always result in better applications. There are known vulnerabilities in both the Java Virtual Machine (JVM) and the CLR environments. In their report, ".NET Security: Lessons Learned and Missed from Java,"8 researchers Paul and Evans discuss a number of the problems found in both the .NET just-in-time (JIT) and the Java bytecode verifiers. One of the compelling reasons to use these verifiers is to perform code checks, which should reduce software faults. If the verifiers fail, but the application development team trusts that they have not failed, the result could be a false sense of security. The problem is not that Sun or Microsoft is doing a bad job with verification. Rather, it is a problem with placing too much trust in a single mechanism; in this case, the verifier.

Code re-use is another area where blind trust in the ability of managed code to deliver complete application security can lead to problems. Because it can be difficult to make blanket decisions about security when writing code, many decisions are made on a context-driven basis. From a business perspective, what may be acceptable risk in a component for an application in its first iteration may not be acceptable in a subsequent iteration. In some instances, this can lead to insecurity in the application later in the cycle. If the component has already passed review and is considered safe, re-using it as the environment around the component changes could introduce risk that was not present before.

While managed code is most certainly a significant market trend in the ways applications are developed, the use of managed code environments is not a replacement for secure design, code scanning, and testing.

Implementation Patterns

In most enterprises, code testing is done by two types of users: developers and auditors. There is much room for additional granularity in these two groups, such as auditors who test source code versus auditors who penetration test applications. The high-level goal of developers is to write code that results in applications or components. Many developers perform the majority of their work inside of an integrated development environment (IDE). For developers who work in an IDE, having code-scanning tools that run within the IDE is useful because it reduces the time and steps required to scan the code.

Stand-alone source code scanners are most commonly used by developers (who do not use an IDE) and auditors. Auditors' high-level goals are different from those of developers. The auditor is responsible for verifying the application and source code against an identified standard and is generally less concerned with speed than with getting accurate results. Due to political boundaries across business units and the rise in outsourcing, the auditor may not have continuous access to code repositories. Auditors may be called on to test delivered source code and then return the results and recommended actions to the development team.

The binary code scanners provide an interesting complement to the source code scanners because binary code scanning does not require source code as input. Runtime analysis tools do not run inside IDEs. These tools are stand-alone products that are used by testing and audit teams.

Code-Scanning Technology and Products

There are a number of vendors in the code-scanning space. This section explains the technical details of the static and runtime analysis tools and provides details about the market leaders.

Static Analysis

Static analysis of code looks at the code in its static, not running, form. Code can be reviewed as source or binary, but the preponderance of available tools focus on source code. One of the reasons for this is because, as noted earlier, the sooner in the development process a problem can be found, the cheaper it is to fix from a time/cost perspective. Once a tool has been compiled into a binary, the cost to fix problems goes up. However, oftentimes the binary is the only thing available for scanning.

Dynamic Runtime Analysis

Executable code can be tested by runtime analysis tools. These tools examine how the program functions and what the application does when it fails. By definition, these tools can only be used after the application source code has been compiled into an executable, so they must be used later than source code scanning in the development process.

Runtime analysis tools can expose problems such as memory corruption errors and leaks, which may indicate faults in the underlying source code. Runtime tools can also complete performance profiling of the application and provide details on execution paths. Some runtime analysis tools provide the ability to graphically represent interactions between application components.

Security Specific Static Analysis Source Code Scanners

The companies described in this section have created tools specifically aimed at finding security flaws.

Fortify

Fortify Software offers the following tools in a suite-based offering:

- Audit Workbench Source Code Analysis
- Red Team Workbench Attack Simulation (for web applications)
- Software Security Manager

The Fortify Analysis Engine can scan C, C++, C#, Java, and JSP. Fortify performs semantic analysis and maps data and control flows. The Audit Workbench knowledgebase combines coding rules, “rule packs” created using Fortify research, customer input, and data from Cigital. Rule packs are updated regularly by Fortify. Customers can also create their own coding rules. The product can be tuned inside the IDE to perform faster checks.

The Red Team Workbench is an attack simulation tool for web applications. Fortify co-created this tool with auditing firm Ernst and Young. Information gathered during the source code analysis can be fed into the Red Team tool for additional insight.

The Software Security Manager can be used by project managers to track the progress of an application or set of applications. Defects can be categorized by type and number of defects and the results of a current scan can be compared to previous results.

Ounce Labs

Ounce Labs' products are part of the Prexis family of solutions for scanning C, C++, Java, and JSP. Ounce Labs concentrates on providing tools for managers and auditors but also has a scanning solution for developers that works inside popular IDEs.

The Prexis components include:

- Prexis/Engine
- Prexis/Insight
- Prexis/Pro

The Prexis/Engine scans source code looking for structural problems such as race conditions and buffer overflows. Ounce uses optimization heuristics rather than formal methods to speed the work of the engine. The Prexis/Engine compares findings against the information knowledge stored in the Prexis Security Knowledgebase. Vulnerabilities are categorized, rated, and assigned what Ounce calls the “V-Density” (vulnerability density) weight, which helps to ascertain the potential severity of the identified problem.

Prexis/Insight is a management dashboard that provides graphic reporting information for consumption by executives and project managers. The auditor/manager focus of the company is clear in this product. Reports are clear, easy to read, and can be configured to tune the presentation of findings in ways that matter to executives (e.g., comparing overall V-density levels across all scanned applications).

Prexis/Pro is a developer remediation tool. Information from the Prexis/Engine is fed into Prexis/Pro, where developers or auditors can drill down on the vulnerability to read about its severity rating and suggested steps on how to remediate. Double-clicking the vulnerability links the Prexis/Pro information to the associated line of code within the IDE so it can be corrected if necessary.

Secure Software

Secure Software takes a combined process and tools approach to the security application problem. In addition to source and binary scanning tools, Secure Software has built the Comprehensive, Lightweight Application Security Process (CLASP), a set of processes and artifacts that enterprises can adopt to help weave security throughout the SDLC. CLASP information and an implementation guide can be found at www.securesoftware.com.

Secure Software's tools in the CodeAssure product line complement the CLASP offering:

- CodeAssure Workbench
- CodeAssure Auditor
- CodeAssure Integrator
- CodeAssure Management Center

The CodeAssure Workbench is a static code analysis checker. The checker uses information stored in the Code Assure Knowledgebase to help contextualize and validate discovered problems. CodeAssure supports C and Java now and will support C++ in late Summer 2005.

CodeAssure Auditor performs binary analysis for vulnerabilities related to checking API calls and coding behaviors. The CodeAssure Integrator is designed to help integrate the information gathered by the analysis engine into the quality assurance process.

All of the CodeAssure products can feed information to the CodeAssure Management Center. The Management Center can cross-compare rated vulnerability levels of applications in the enterprise. It also supports rules-based checking on configurable policy requirements.

TQM Static Analysis Source Code Scanners

The vendors in this section are a little more established than the security-specific source code scanners. These companies are focused on finding many types of in software, not just security issues.

Coverity

Coverity offers a static code analysis tool for C and C++ known as Prevent. Coverity's focus is on software quality, including security checks. Prevent uses the Coverity Analysis Engine to run the scans. Coverity Prevent integrates with a number of popular compilers, including GCC (GNU Compiler Collection), Microsoft Visual Studio, and Sun cc.

GrammaTech

GrammaTech offers the following static analysis tools for C, C++, and Ada:

- CodeSonar
- CodeSurfer

CodeSonar is a static analysis tool that looks for flaws such as null pointer errors and buffer overflows. CodeSonar supports GCC, Microsoft Visual Studio, and Sun cc compilers. GrammaTech also has a stand-alone code-scanning tool, CodeSurfer, which provides a browser-based, graphical view of source code.

Klocwork

Klocwork offers four tools:

- inSpect
- inSight
- inTellect
- inForce

inSpect is a static analysis code scanner for C, C++, and Java. inSpect looks for defects and architectural issues as well as potential security problems. inForce runs inside IDEs (such as Eclipse and Microsoft Visual Studio/.NET) and checks for defects, opportunities for code optimization, and security. inTellect is the management console where a number of defects can be tracked and viewed over the course of the application lifecycle.

Parasoft

Parasoft has a comprehensive line of unit testing and scanning tools for developers. Its products include:

- Jtest
- C++Test
- Group Reporting System (GRS)

Both Jtest and C++Test work inside the developer's IDE. Java supports IBM WebSphere, Eclipse, JBuilder, and Sun ONE Studio. C++ supports Microsoft's Visual Studio and .NET, and compilers GCC for Windows and Sun Forte C++ for Solaris. Information gathered during scans can be sent to the GRS, which can help track the progress of the application development team.

Security-Specific Binary Code Scanners

For C, C++, and Java binaries, there are few commercially available security-specific binary code-scanning products. One of the most well established is Logiscan (formerly BugScan).

LogicLibrary

The LogicLibrary Logiscan tool works on Intel x86 binaries compiled from C, C++, and Java code, and on Java 2, Enterprise Edition (J2EE) binaries for SPARC (Solaris and Linux) and MIPS (Windows CE and Linux). Logiscan also integrates with the Eclipse and Rational Application Developer IDEs. LogicLibrary also offers an AppExplorer tool, which provides developers with a 3D, high-level visualization of the application.

Runtime Analysis

Runtime analysis tools check for failures in the application software that occur at runtime.

IBM

IBM's Rational tools include two runtime analysis scanners—PurifyPlus for Windows and PurifyPlus for UNIX—that can be used to check for memory leaks and overall application performance. The products will work with unmanaged languages, C, C++, Java, C#, and VB.NET. Both products can be purchased in a bundle as PurifyPlus Enterprise Edition.

Parasoft

In addition to code-scanning tools, Parasoft has Insure++, a runtime analysis tool for C and C++ application testing. Insure++ can help detect memory leaks, pointer errors, and memory allocation errors.

Software Security Consultants

In addition to the products listed previously, enterprises can also work with consulting firms that specialize in creating and security testing applications. These firms can also provide training for developers and architects to help them understand how to create more secure applications.

Cigital

Cigital is a consulting services group that was founded in 1992. Cigital does not provide tools; it provides consulting to help companies improve their software development process. Cigital consultants can work with enterprises to help improve the security of the code by reviewing development artifacts and scanning code.

Security Innovation

Security Innovation provides consulting, testing, and training services to help enterprises improve the security of their applications. Consultants use manual review and automated scanning to uncover potential flaws. Their training services help developers with aspects of secure programming.

Conclusion

Integrating security into the software development lifecycle (SDLC) will uncover security flaws earlier in the process and should result in more robust, less vulnerable applications. Although there is no replacement for security-aware design and a methodical approach to creating more secure applications, code-scanning tools are a very useful addition to the process. Enterprises that understand how to create more secure applications can benefit from greater efficiencies in the development process, and they may find that by creating more secure applications, there is less need for post-production security software whose entire function is to protect and patch insecure software.

Notes

1 Barry W. Boehm, Philip N. Papaccio. “Understanding and Controlling Software Costs.”*IEEE Transactions on Software Engineering*14:10. Oct 1988. 1462-1477.

2 “The Economic Impacts of Inadequate Infrastructure for Software Testing.” *National Institute of Standards and Technology (NIST)*. May 2002. www.nist.gov/director/prog-ofc/report02-3.pdf.

3 Hayashi. “Rough Sailing for Smart Ships.” *Scientific American*. Nov 1998. http://www.sciamdigital.com/browse.cfm?ITEMIDCHAR=5D2AED73-433D-48A4-85F1-9ABF90A2127&methodnameCHAR=&interfacenameCHAR=browse.cfm&ISSUEID_CHAR=95C97E8D-4E3A-4EE5-A478-944FC3A850A&ArticleTypeSubInclude_BIT=0&sequencenameCHAR=itemP.

4 J.L. Lions. “ARIANE 5 Flight 501 Failure: Report by the Inquiry Board.” 19 Jul 1996. http://homepages.inf.ed.ac.uk/perdita/Book/ariane5rep.html. Perdita Stevens, Rob Pooley. *Using UML: Software Engineering with Objects and Components*. UK: Addison-Wesley, 1999.

5 Sam Williams. *Free as in Freedom: Richard Stallman's Crusade for Free Software*. CA: O'Reilly and Associates, Inc., 2002. http://www.oreilly.com/openbook/freedom/.

6 Forrest Shulle, Roseanne Tesoriero. “What We Have Learned about Fighting Defects: Results of the METRICS02 Workshop.” http://www.cebase.org/www/frames.html?/www/researchActivities/defectReduction/non-eWorkshop/what_we_have_learned_about_fight.asp.

7 Barry W. Boehm. “A Spiral Model of Software Development and Enhancement.” *ACM SIGSOFT Software Engineering Notes* 11:4. Aug 1986. 14-24.

8 Nathanael Paul, David Evans. “.NET Security: Lessons Learned and Missed from Java.” *University of VirginiaDepartment of Computer Science*. Dec 2004. http://www.cs.virginia.edu/~evans/pubs/acsac-packaged.pdf.

Related Research and Recommended Reading

Computer Emergency Readiness Team (CERT) database: http://www.cert.org/nav/index_main.html

Joseph M. Juran information: http://www.juran.com/brand.cfm?target=&article_id=21

Microsoft Security Developer Center: http://msdn.microsoft.com/security/

National Aeronautics and Space Administration (NASA) Software Assurance Guidebook and Standard: http://satc.gsfc.nasa.gov/assure/assurepage.html

National Institute of Standards and Technology (NIST) High Integrity Software Systems Assurance website: http://hissa.nist.gov

National Science Foundation (NSF)'s Center for Empirically Based Software Engineering (CeBase) website: http://www.cebase.org/

The Open Web Application Security Project website: http://www.owasp.org/index.jsp

The "Trustworthy Computing" memo: http://www.news.com/2102-1001_3-817210.html?tag=st.util.print

"Vilfredo Pareto" entry in *Wikipedia*: http://en.wikipedia.org/wiki/Vilfredo_Pareto

Author Bio

Diana Kelley

Vice President and Service Director

Emphasis: Information security, compliance, policy and risk management, software and application security, web application firewalls, collaboration security, instant messaging security, security information and event management, network security architectures, and host intrusion prevention

Background: Over 16 years of experience creating secure network architectures and business solutions for large corporations. Previous positions include executive security advisory for CA; manager for KPMG's Financial Services Consulting practice; vice president of security technology for Safe3W (acquired by iPass); senior analyst for Hurwitz Group; general manager of a development group at Symantec Corp.

Primary Distinctions: Co-author, with Ed Moyle, of "Cryptographic Libraries for Developers." Speaks at major conferences such as RSA, ISSA, Information Security Decisions, InfoSec World, and N+I. Chaired MISTI's Mobile and Wireless Security conferences (2003-2005) and Identity Management conferences (2006-2007). Quoted as a security expert in publications such as Information Security Magazine and The Wall Street Journal. Contributes articles to publications such as TechTarget, ComputerWorld, SC Magazine, and CNET.