



Whitebox Security Testing Using Code Scanning

A WHITE PAPER PROVIDED TO ASPE BY SECURITY INNOVATION

Whitebox Security Testing Using Code Scanning

written for Dr. Dobb's Journal

Joe Basirico

Security Analyst
Security Innovation

 **ASPE**
SDLC TRAINING **Presents...**



PREPARED BY
SECURITY INNOVATION[®]
THE APPLICATION SECURITY COMPANY

Security Innovation, Inc.
187 Ballardvale Street, Suite A170
Wilmington, MA 01887
+1.978.694.1008
www.securityinnovation.com



Whitebox testing is notoriously difficult to do. Without automatic code scanning tools, scanning the source code requires a keen eye, concentration and an enormous amount of time to scan each line for security vulnerabilities. As hackers become more sophisticated at finding security vulnerabilities and writing exploitative code, it becomes more necessary to take every precaution before shipping software. These precautions can range from security training throughout the security development lifecycle (SDLC) to using tools such as source code scanners and vulnerability scanners.

Last year, more vulnerabilities were reported in shipping software than any previous year¹. Nearly 6,000 new vulnerabilities surfaced in shipped software, and the state of the industry seems grim. Despite attempts by the larger and more proactive players in the industry, like Microsoft, IBM, HP and CISCO, to incorporate security in all phases of the SDLC, the number of reported vulnerabilities continues to increase. Where are the greatest weaknesses in software security? Were nearly two thousand more vulnerabilities discovered because security trainers did not transfer their knowledge properly? Or are hackers getting better at discovering these vulnerabilities? Do software corporations rely too much on perimeter defenses, such as firewalls, intrusion detection systems, deep packet inspectors and anti virus solutions? Is it just sheer complacency? Or is it really a combination of all of these things and more?

The final, and most important question, is one we can all think about and help answer: What can we do next to ensure that we ship fewer and fewer security vulnerabilities in the subsequent versions of our software?

The answer is to train every member of the team to think about security in all phases of the SDLC. From the moment the Project Managers are talking to customers and gathering requirements to the final shipping build and deployment, security must remain in focus. This is a difficult task to accomplish considering that most software teams are already stretched thin with more demands on their time than hours in the day. Tightened budgets for development and testing teams make purchasing new tools, even when legitimate, very difficult.

One tool that is invaluable in the development phase of the SDLC is the static analyzer. This tool can discover elusive programming errors before run time when they become much more difficult to find. Static analysis tools can help you discover many logical and security errors in an application before compilation. When choosing static analysis tools, consider the benefits and limitations and what types of bugs they are good at finding.

Successes with Static Analysis Tools

Microsoft has learned how to make static analysis tools work to its advantage. Every major project at Microsoft goes through rigorous testing with static analysis tools. Microsoft employs these tools both at check-in time and at the final build. Each code modification is checked on the developer's computer and in the context of the complete solution before the nightly build. Microsoft has developed its own internal static analysis tools called Prefix and Prefast. These tools have evolved over the years and are quite adept at identifying both security flaws and functional problems before build time. The ability to run static analysis on all of the Windows source code is a testament to how mature these tools have become.

NASA also requires that every code change to mission-critical applications go through static analysis. Every warning and suggestion the analyzer finds must be either fixed, or a comment must be added to the source code explaining why the warning or suggestion required no change to the code.

Most major software companies employ some form of static analysis tool; in fact, it is difficult to find an example where the exercise of these tools has not helped the software company deliver a more robust product at ship time. These tools can help discover obscure vulnerabilities in source code and ensure that a more secure application gets to the customer on time.

¹ Source: CERT/CC

State of the Field

The field of static analysis tools is becoming heavily populated and very competitive. Companies like Klocwork, Fortify and Compuware have created some very interesting and diverse offerings for static analysis. The techniques used by static analysis tools are often guarded as the key intellectual property of the company; simple pattern matching is not enough.

Some of the techniques employed by static analysis tools are:

- ▶ Semantic checking
- ▶ Strong type checking
- ▶ Memory allocation checking
- ▶ Logical statement checking
- ▶ Interface and include problem checking
- ▶ Security checking
- ▶ Metrics
- ▶ Simulation

Should we call out each tool before the respective paragraphs to allow the reader to fulfill the expectation of covering each of the tools listed above? You could include examples of each kind of tool at the end of each description.

Semantic analysis allows the analyzer to discover the basic structure and relation of each method within the application. The static analysis engine can then build a syntax tree that the simulator will use later to calculate how the application will execute at runtime. This allows the static analyzer to find bugs deeper in the application by traversing the syntax tree for method context that approximates what would occur at runtime.

Strong type checking helps prevent dangerous type casting assumptions such as trying to cast a decimal to an integer and losing precision at runtime. This can cause many security vulnerabilities in memory allocation and rounding. Remember the adage one plus one may equal 3 for very large values of 1.

Within the strong type checks the static analysis engine also ensures that each variable is properly initialized before use. Since it can quickly discover how and when the variable is to be used, it can easily warn the developer of cases when uninitialized variables could cause problems. These problems often manifest themselves as "divide by zero" errors or other improper function calls.

Many modern static analysis tools attempt to match *memory allocations* with deallocations. This helps uncover many memory leak problems where a developer forgets to deallocate small bits of memory which, over time, may build up and cause the system to become unstable or develop a security flaw. Memory leaks are notoriously difficult to find at runtime, and can require many hours of repetitive, pedantic testing.

Logical statement checking can help the developer or whitebox tester to discover cases in which an 'if' or 'while' statement may always evaluate to the same result. Often logical statements can be built up, version after version, until simply getting one correct output is enough. Developers sometimes get it working then step slowly away, trying not to touch their 'if' statement with seven 'ands' and five 'ors.' These under-tested 'if' statements may have fallen into the case when no matter what inputs are given it will always result in the same outcome. Static analysis tools can quickly evaluate these statements by simulating all possible values of each variable and discover which logical statements should be redesigned.

Security exposure may be reduced by removing unnecessary libraries from the application. Static analysis tools can quickly check which libraries or APIs are required by the application and alert the developer to remove them. Since we inherit the security vulnerabilities from each of the libraries we call into, removing unnecessary libraries is a security best practice. There have been many cases of developers and testers

failing to test for buffer overflows in their application because they thought they were safe due to the use of a managed application written in .Net or Java. Many of these applications call into legacy code that was written long ago and contains other security vulnerabilities; removing all unnecessary libraries can help lessen the attack surface.

Many other *security checks* can be found at compile time. Some system API functions have been deprecated and should be replaced with their safer cousins. A static analysis tool can quickly discover the use of any of these functions and notify the developer of a safer replacement. Possible buffer overflow conditions can be predicted so the developer can fix them before they are discovered and exploited through the application. Discovering buffer overflows at this stage also shows the developer the exact line and conditions when a buffer overflow may occur which can be very helpful in finding and fixing the vulnerability.

Time-of-check versus time-of-use problems can be discovered as well by showing places where the developer has checked the availability of a resource and has allowed a significant amount of time to pass before using the resource. Time-of-check versus time of use errors can allow escalation of privilege attacks which may allow tampering of a resource to which the user shouldn't have access or information disclosure.

Metrics can help a developer understand where there is unnecessary complexity in a piece of code. This can help increase performance of the software significantly, which will lessen the possibility of a denial of service attack. Static analysis tools can predict which functions are very complex using Cyclomatic complexity, Functional file coupling and Functional file cohesion. Other metrics, such as ratio of commented lines to source lines, can point to places in the source where the code may need to be commented more effectively.

The *simulation engine* is one of the most powerful and most guarded pieces of the static analysis tool. The simulation engine allows the tool to predict how the application will behave after it has been compiled and run on the system. In a simulation engine, the simulator selects a function, generates data for each of the variables, and runs the function through every possible code path. If information about proper data ranges for those variables exist within source code they will be used, if no information exists then random values selected between the max and min for that data-type are used. Testing with a static analysis simulation engine is a good way to get around the inherent shortcomings of traditional static analysis tools.

The final check a static analysis tool can do is to crawl the source code to map out every possible code path and discover unused or unreachable code. These dead code pockets are orphaned and unreachable so they remain untested throughout the product cycle. In a future release, bug fix or code modification, the dead code may become active, thereby exposing untested security vulnerabilities or other functional flaws. Under certain circumstances a skilled hacker may even be able to circumvent the built in constructs within an application and execute the untested code which then may contain exploitable security flaws.

Benefits

Employing static analysis tools during testing can drastically reduce the number of bugs which often make their way to the blackbox testing phase. These tools can be executed very quickly, and the scanning is very easy to do before source code check-in.

While not often considered, one of the first things a hacker will attempt to do when compromising a system is to steal source code. Source code will allow the hacker to discover vulnerabilities in the application at the source level very quickly by using security scanning tools. This low hanging vulnerability should be (?)eliminated before ship or it may open a large attack surface if the source is compromised. With the gaining prevalence of Java and .Net applications, decompilers also give the attacker a significant edge to discover vulnerabilities with source code scanners.

Scanning tools can help reduce development and testing costs because they catch problems early in the SDLC. Bugs are expensive to fix later in the lifecycle of software, and security vulnerabilities are infamous for being elusive. Source code scanners are great assets because they discover vulnerabilities that are difficult or time consuming to find by other means, including:

- **Buffer overruns** – These vulnerabilities are eye candy to a hacker. They are often exploitable and may allow the hacker to take complete control over the system. Buffer overflow conditions are difficult to spot in manual code inspection because multiple conditions may be required to trigger the error.
- **Least privileges** – A process should always run with least privileges, ensuring that if the code is exploited the undesirable code is given limited power.
- **Dangerous Functions** – Some system functions may open your application to possible security flaws; these should be examined and replaced with more secure functions (e.g., replace `strcpy` with `strncpy`).
- **DACL Problems** – A null DACL gives no protection and is a warning sign that an object being used by the application is not as secure as it should be.
- **Canonicalization Problems** – There are many different ways to represent a file, URL or device. A hacker may be able to gain access to a protected file by using alternate representations of the filename.
- **Exception Handling** – If an exception handler is not present, the application may terminate or to be left in an unpredictable state when problems occur.
- **Format String Problems** – Functions such as `printf`, `scanf`, `sprintf` and others may open an application up for problems when user input is interpreted as the format string. This can lead to an exploitable vulnerability which may allow the attacker to execute code remotely.
- **Input Validation** – A hacker may be able to cause complete system compromise if improper input exposes a buffer overrun or format string bug.
- **Ignored Return Values** – Ignoring return values can result in a variety of reliability and security bugs that can be quite hard to debug and reproduce.
- **Package Insertion** – Package insertion can allow un-trusted code to run in the context of a trusted Java application and may therefore spoof or otherwise attack the user.
- **SQL Injection** – SQL injection is a technique used by hackers to probe databases, bypass authorization, execute multiple SQL statements and call built-in stored procedures.

Each of these vulnerabilities must be scanned for and discovered. These are some of the most popular vulnerability types for hackers to attack.

Limitations

No single tool or practice can possibly discover or fix all security vulnerabilities. Multiple tools, training and best practices must be employed instead to ensure that software is as secure as possible. Some of the limitations of static analysis tools include false positives, vulnerabilities that only show up in the environment and a false sense of security.

False positives can be very difficult to weed through and often require significant security experience to discern which warning should be fixed. Fixing every warning reported by a static analysis tool can cause unnecessary code churn, which may lead to the introduction of more functional and security flaws. False positives that are not easily interpretable can overwhelm a developer to the point that he or she becomes disenchanted with the tool. This plays into the psychological acceptability of the tool; if it is too difficult or causes significantly more work for the developer or tester to use, it may be discarded, despite the potential of increased security in the end product.

Many vulnerabilities occur only in specific environments and will only be discoverable on certain install beds. These vulnerabilities are often caused by low system resources, library and API versioning problems, and insecure settings. These types of vulnerabilities are impossible to detect before runtime.

Static analysis tools attempt to simulate resource contention and some versioning problems, but the vulnerability can only be detected at runtime with dynamic analysis tools such as a debugger or fault injection tool. Library and API versioning problems are difficult to detect and test for before runtime; however, with strong naming conventions, a developer can take precautions to ensure a certain library has been installed on the system. This approach, while secure, is sometimes unused because testing on every available system is difficult, costly and limits the number of compatible systems the application can be installed on.

Static analysis tools can not foresee possible insecure deployment settings and thereby miss this significant attack surface. Ensuring the product ships securely and is installed securely by default is challenging and must be outlined in the requirements from early on. It is essential to understand:

- ▶ The environment your software will be deployed in,
- ▶ How your users will use your software
- ▶ How knowledgeable the users are, and
- ▶ How much configuration is likely to occur after the product has been installed on the system.

Once these questions have been answered, secure requirements can be written to help keep your users safe in many deployment situations.

When to Use Static Analysis Tools

Since static analysis tools are leveraged before build time, they can save time, money and re-testing aggravation. The cost to find and fix a bug rises drastically throughout the product cycle: if a bug is found by the customer after the application has been released, it can cost literally millions of dollars. Since static analysis tools are run early in the development cycle and can point out the specific line number where a programming error has been made, the bug can be found and fixed very quickly and inexpensively. Advanced static analysis tools not only show the line number of the bug but also display the code paths that lead to execution of the bug.

Static analysis tools can and should be used throughout the product cycle. Developers can use a light weight version of the tool to check for simple bugs that may have been missed at development time. Build managers or lab technicians should use the tool to isolate the more sophisticated bugs at code integration time. After a build is created, testers can use static analysis tools to ensure code coverage and to discover complex sections of the product that should be tested more thoroughly.

Developers can run a stripped down version of the static analyzer on their local build machine before check-in to enforce coding standards, readability and check for security bugs. Enforcing coding standards and readability will ensure that code is easy to update and maintain in future releases. Many security bugs like buffer overruns, input encoding issues and use of dangerous functions can be found before the source is built into the complete shared source tree.

Once the developers check their code changes into the source tree, the build manager can run the static analyzer to check for integration bugs. In this step, the build manager can enable options that may not have been available to the developer, such as simulations, function integration options, and dependency diagrams. Simulations may require some external states to be set before running the newly added code which will only be available with the complete source tree. At this point, the build manager can send bug reports back to the developers so they can fix code integration errors before they even make it to the build. The two sided analysis, of developer and build manager, will help remove many bugs before the source is built, keeping development and bug fix costs significantly down.

After the build is complete, testers can use static analyzers to discover areas of higher cyclomatic complexity which will provide insight to where deeply hidden bugs may lie. Using the metrics and complexity features of a static analysis tool, testers can discover code paths to execute the complex

function at runtime. Once the code path has been found, the tester can use that information to execute the complex function from within the built application and discover complex runtime errors.

Good Attributes to Watch Out for

There are many great static analysis tools available on the market today. Many of them offer great features and promise to secure your application from unique and isolated threats. Matching the static analysis tool to your specific development process can mean the difference between a painful process and a seamless integration with your current development process.

The most obvious requirement for any static analysis tool is to make sure it integrates well with your current development environment. Whether it's a plug-in for Eclipse or an add-in for Visual Studio, make sure that it works well and does not require too much hardware overhead such as extra processor power or memory. It's no fun to be forced to upgrade all the development machines in order to get a new static analysis tool to work. Also, make sure that the static analysis tool you choose supports not only the programming languages you are using currently, but any anticipated languages.

Many static analysis tools are definition based, which allows you to update them as new vulnerabilities are discovered. This is a great feature which can save you a lot of time and money down the road. Also look for a static analysis tool that will allow you to write your own definitions. This will allow for even tighter coupling between the base static analysis tool and your internal best practices.

Ask your current development team if they have suggestions for static analysis tools that they've found to be effective. Ramp-up time on these tools can be significant so if there is a developer who is already experienced with a tool it may be worthwhile to use that developer as a resource when deploying the tool. If you do not gravitate towards a specific static analysis tool, consider the overhead time it takes to train the development team on each tool and the support that will be provided by the static analysis tool company.

Before Deploying a Static Analysis Tool

Before diving into the use of static analysis tools, consider deploying a few on small projects to get a feel for their usability and ability to integrate with tools already being used in your development shop. These small projects can range from small mockup projects specifically designed to test the tool to testing the whole project for a short amount of time. If you choose the latter, understand that there will be a lot of overhead time as the developers learn a new tool.

When deploying a new tool, be sure to appoint a dedicated team member who will master the tool. This person should understand all the features of the tool and how to apply them to the current project and be a heavily trained and competent developer.

Key Advice – Security (Not Tool) Knowledge is King

For static analysis to be most effective, it must be coupled with security awareness training for developers so they can understand the threats and readily interpret and act on the results. There are a number of secure coding courses, which range from classes which assume no previous security knowledge to advanced security courses which challenge developers to understand secure development practices completely. However, make sure the source of your training is well-qualified and knowledgeable – there are many self-styled experts ready to pass themselves off as application development, testing and security gurus.

Remember, too, once you receive training and feel knowledgeable in the field of application security, make sure to tap into the services of external security testers to help audit and gauge your performance. Keep security in the front of your mind throughout the development process to ensure more secure products that will delight your customers.

About the Author

Joe has spent most the majority professional career studying security and developing tools that assist in the discovery of security vulnerabilities and general application problems. His primary responsibility at Security Innovation is to deliver security courses to Software Teams in need of application security expertise. He has trained developers and testers from numerous world-class organizations such as Microsoft, HP, EMC, Symantec, Liberty Mutual, Sony, State Farm, Credit Suisse, Amazon.com, Adobe and ING. Joe is also responsible for participating in customer security process assessments as well as security engineering activities such as security design reviews, security code reviews, and security testing and security deployment reviews.

Joe has been interviewed on several occasions by media outlets including SC Magazine and Software Test & Performance. He has written several whitepapers and articles that focus on vulnerabilities at the source code level for *CIO Update*, *Dr. Dobbs Journal* and *DM Review*. Joe is a seasoned practitioner and researcher in the field of incorporating security into the SDLC and a highly demanded presenter and for the topic of software development best practices. He has delivered presentations at several world-class venues, including Software Test & Performance, Software Security Summit, Microsoft PDC and Compuware OJ. X

Joe holds a B.S in Computer Science Graduate from Montana State University.

About Security Innovation

Security Innovation is the authority on application security and a leading provider of risk analysis, risk mitigation and education services to mid-size and Fortune 500 companies. Global technology vendors and enterprise IT organizations rely on our services to identify security risks in their software systems and development processes and facilitate the changes needed to mitigate them. The company is headquartered in Wilmington, MA and has offices in Seattle, WA and Amsterdam, the Netherlands.