More Complex = Less Secure

Miss a Test Path and You Could Get Hacked

"The future of digital systems is complexity, and complexity is the worst enemy of security." *Bruce Schneier, Founder and CTO, Counterpane Internet Security, Inc. 2000*

About this Paper

Software systems are becoming less secure even as security technologies improve. There are many reasons for this seemingly paradoxical phenomenon, but they can all be traced back to the problem of complexity.

- Complex systems have more lines of code and therefore security bugs.
- Complex systems have more interactions and therefore more security bugs.
- Complex systems are harder to test and therefore are more likely to have untested portions.
- Complex systems are harder to design securely, implement securely, configure securely and use securely.
- Complex systems are harder for users to understand.[1]

This paper will show you how using software complexity metrics, measuring control flow integrity, and performing sneak path analysis help you make your applications more secure than previously thought possible.

Security Debuggers vs. Security Testing

Tools that search for known exploits are analogous to debuggers and are employed using a reactive model rather than a proactive one. Many exploits deal with interactions: interactions between code statements, interactions between data and control flow, interactions between modules, interactions between your codebase and library routines, and interactions between your code and attack surface modules. This is why cyclomatic complexity path and subtree analysis is an important complementary technique. Being cognizant of paths and subtrees within code is crucial for determining sneak paths, performing impact analysis, and testing to verify control flow integrity. It is crucial that both security debuggers and security control flow integrity test tools are included in your arsenal.

Source Code Analysis vs. Binary Analysis

As is the case with static analysis and dynamic analysis, the two approaches of source and binary analysis are complementary. Source analysis is platform (architecture and operating system) independent, but language-specific; binary analysis is more language-independent but platform-specific. Source code analysis has access to high-level information, which can make it more powerful; dually, binary analysis has access to low-level information (such as the results of register allocation) that is required for some tasks.

Bottom line is: The binary approach effectively analyzes what the compiler produces, whereas the source approach effectively analyzes what the developer produces.It is true that binary (compiled) code represents the actual attack surface for a malicious hacker exploiting software from the outside. It is also true that source code analysis has differentiated itself in a complementary way by finding the enemy within software development shops. There have been studies indicating that exploits from within are far more costly than those from the outside. Source code analysis can be employed much earlier in the software development lifecycle (SDLC). Libraries and APIs can be tested early and independently of the rest of the system. Binary analysis requires that at least an entire executable, if not an entire subsystem or system is completed.

In binary analysis, it is true that white box analysis reporting can be generated. However, these reports are indirect, and do not always correlate exactly back to the source code logic; therefore, detailed analysis may be more difficult than humans analyzing source code analysis reporting. Furthermore, compilers and their options (such as optimization) can cause the correlation between binary analysis reporting and source code to be even more different.

Security Problems as Software Grows More Complex

As software grows more complex, it contains many more flaws for hackers to exploit. Powerful computer systems and increasingly complex code will be a growing cause of insecure networks. We are getting these great performance improvements, which leads to increases in complexity. Today, nobody has any clue what is running on their computer.

In the *Final Report of the Defense Science Board Task Force on Mission Impact of Foreign Influence DOD Software - November 2007*, the following statements were made: The complexity of software itself can make corruption hard to detect. Software has been growing in the dimensions of size, complexity and interconnectedness, each of which exacerbates the difficulties of assurance. Software complexity is growing rapidly and offers increasing challenges to those who must understand it, so it comes to no surprise that software occasionally behaves in unexpected, sometimes undesirable ways. The vast complexity of much commercial software is such that it could take months or even years to understand. The Nation's defense is dependent upon software that is growing exponentially in size and complexity. The following findings were found in this report: The enormous functionality and complexity of IT makes it easy to exploit and hard to defend, resulting in a target that can be expected to be exploited by sophisticated nation-state adversaries. The growing complexity to the microelectronics and software within its critical systems and networks makes DoDs current test and evaluation capabilities unequal to the task of discovering unintentional vulnerabilities, let alone malicious constructs.

One of the key properties that works against strong security is complexity. Complex systems can have backdoors and Trojan code implanted that is more difficult to find because of complexity. Complex operations tend to have more failure modes. Complex operations may also have longer windows where race conditions can be exploited. Complex code also tends to be bigger than simple code, and that means more opportunity for accidents, omissions and manifestation of code errors.

The central enemy of reliability is complexity. Complex systems tend to not be entirely understood by anyone. If no one can understand more than a fraction of a complex system, then, no one can predict all the ways that system could be compromised by an attacker. Prevention of insecure operating modes in complex systems is difficult to do well and impossible to do cheaply. The defender has to counter all possible attacks; the attacker only has to find one unblocked means of attack. As complexity grows, it becomes ever more natural to simply assert that a system or product is secure as it becomes less and less possible to actually provide security in the face of complexity. Despite a wealth of testing tools that claim to catch bugs, the complexity of software makes security flaws and errors nearly unavoidable and increasingly common.

The complexity explosion in software is exponential. The challenges of rising system complexity for software developers cannot be overstated. There is a movement to more complex systems, and the operating system is forced to take on a larger role in managing that complexity. We have passed a critical juncture where a new paradigm is required. You get to a certain size of the software where your odds of getting a really serious error are too high. We have to change the whole rules of engagement. In the

1970s, the average car had 100,000 lines of source code. Today it's more than a million lines, and it will be 100 million lines of code by 2010. The difference between a million lines of code and 100 million lines of code definitely changes your life.

McCabe Complexity Metrics

Certain characteristics of computer software make it more- or less- vulnerable. Complexity drives insecurity. *Cyclomatic complexity* is the most widely used member of a class of static software metrics. *Cyclomatic complexity* may be considered a broad measure of soundness and confidence for a program. Introduced by Thomas McCabe in 1976, it measures the number of linearly-independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs. *Cyclomatic complexity* is often referred to simply as program complexity, or as McCabe's complexity. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format.

The higher the complexity the more likely there are bugs. The more bugs the more security flaws. A certain number of "complexity bugs" can be found through programmer vigilance. Get to know your code. Get to know how the pieces work and how they talk to each other. The more broad a view you have of the system being programmed, the more likely you will catch those pieces of the puzzle that don't quite fit together, or spot the place a method on some object is being called for some purpose it might not be fully suited.

There is an absolute need to examine code for security flaws from this position of knowledge. When considering security-related bugs, we have to ensure the system is security proof against someone who knows how it works, and is deliberately trying to break it, again an order of magnitude harder to protect against than a user who might only occasionally stumble across the "wrong way to do things".

The McCabe Complexity Metrics can be used to quantify security exposure and impact and can be used to unravel previously incomprehensible logic, design and sneak paths.

Cyclomatic Complexity

Cyclomatic complexity v(G) is a measure of the logical complexity of a module and the minimum effort necessary to qualify a module. Cyclomatic complexity is the number of linearly independent paths and, consequently, the minimum number of paths that one should (theoretically) test. It

- Quantifies the logical complexity
- Measures the minimum effort for testing
- Guides the testing process
- Is useful for finding sneak paths within the logic
- Aids in verifying the integrity of control flow
- Can be used to test the interactions between code constructs

Module Design Complexity

Module design complexity iv(G) of a module is a measure of the decision structure which controls the invocation of the module's immediate subordinate modules. It is a quantification of the testing effort of a module as it calls its subordinates.

The module design complexity is calculated as the cyclomatic complexity of the reduced graph. Reduction is completed by removing decisions and nodes that do not impact the calling control of the module over its subordinates. Design complexities exist because:

- Modules do not exist in isolation
- Modules call child modules
- Modules depend on services provided by other modules

This metric quantifies the interaction of modules with subordinates under security review. How much security testing is required to integrate this module into the rest of the system?

Module Global Data Complexity

Global data complexity gdv(G) quantifies the complexity of a module's structure as it relates to global and parameter data. Global data is data that can be accessed by multiple modules. This metric can show how dependent a module is on external data and is a measure of the testing effort with respect to global data. Global data complexity also measures the contribution of each module to the system's data coupling, which can pinpoint potential maintenance problems. This metric:

- Isolates the modules with highest external data coupling.
- Combines control flow and data analysis to give a more comprehensive view of software than either measure would give individually.

Module Specified Data Complexity

Specified data complexity sdv(G) quantifies the complexity of a module's structure as it relates to user-specified data. It is a measure of the testing effort with respect to specific data. A data dictionary is used to select a single data element, all elements with a specific data type, or a variety of other selection criteria. The specified data complexity then quantifies the interaction of that data set with each module's control structure. It indicates the data complexity of a module with respect to a specified set of data elements, and equals the number of basis paths that you need to run to test all uses of that specified set of data in a module. It allows users to customize complexity measurement for data-driven analyses

For example*, specified data complexity* can be used to analyze the complexity, context and testing effort of the four Standard Windows Socket Routines: Which modules are using recv() (TCP), recvfrom() (UDP), WSARecv() (TCP) and WSARecvFrom() (UDP).

Actual Complexity

Actual complexity ac of a module is defined as the number of linearly independent paths that have been executed during testing, or more formally as the rank of the set of paths that have been executed during testing. The structured testing criterion requires that the actual complexity equal the cyclomatic complexity after testing. The actual complexity is a property of both the module and the testing. For example, each new independent test increases the actual complexity. Actual Complexity should be used to measure code coverage on attack surface modules.

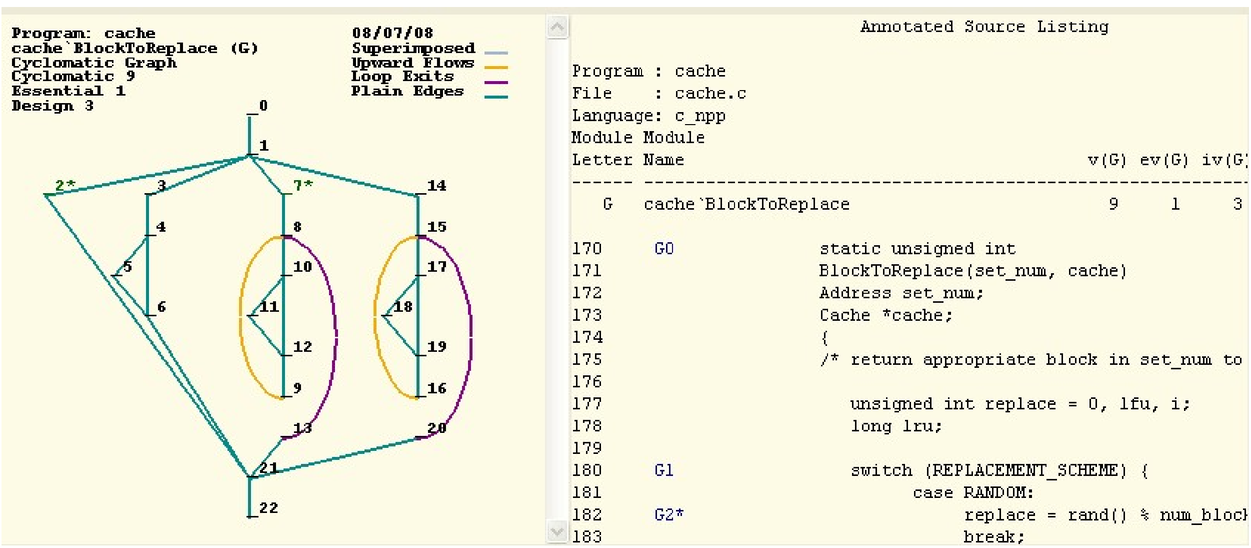
Design Complexity

Design complexity, S0, of a design G, is a measure of the decision structure which controls the invocation of modules within the design. It is a quantification of the testing effort of all calls in the design, starting with the top module, trickling down through subordinates and exiting through the top. It is a measure of the complexity of a design. It is used to identify the minimum yet effective integration tests. This metric can also help assessing sneak path subtrees and the impact on the rest of the system.

Measuring Control Flow Integrity

Given the ingenuity of would-be attackers and the wealth of current and undiscovered software vulnerabilities it is important that control flow integrity be verified to ensure strong guarantees against powerful adversaries. A Control-Flow Integrity security policy dictates that software execution must follow a path of a Control-Flow Graph determined ahead of time. The Control Flow Graph in question can be defined by source code analysis and/or execution profiling. A security policy is of limited value without an attack model.Security Analysis without a control and data flow diagram of logic and design is like doing security analysis of a house without schematics, such as a flooring plan or circuitry diagram. Only scanning for known exploits without verifying control flow integrity is comparable to a security expert explaining the obvious, such as windows are open and doors are unlocked, and being completely oblivious to the fact that there is a trap door in your basement. Those insecure doors and windows are only the low hanging fruit.

Module Flowgraphs can be used to understand the algorithms and their interactions. Visualizing logic using flowgraphs can be used for code comprehension, test derivation, identification of module interactions and for sneak path analysis.



Analysis of the module control flow diagram identifies ways that sources could combine with targets to cause problems. A *Cyclomatic Complexity* measurement of ten (10) means that 10 Minimum Tests will:

- Cover All the Code
- Test Decision Logic
- Test the Interaction between Code Constructs

Structural & Attack Surface Analysis

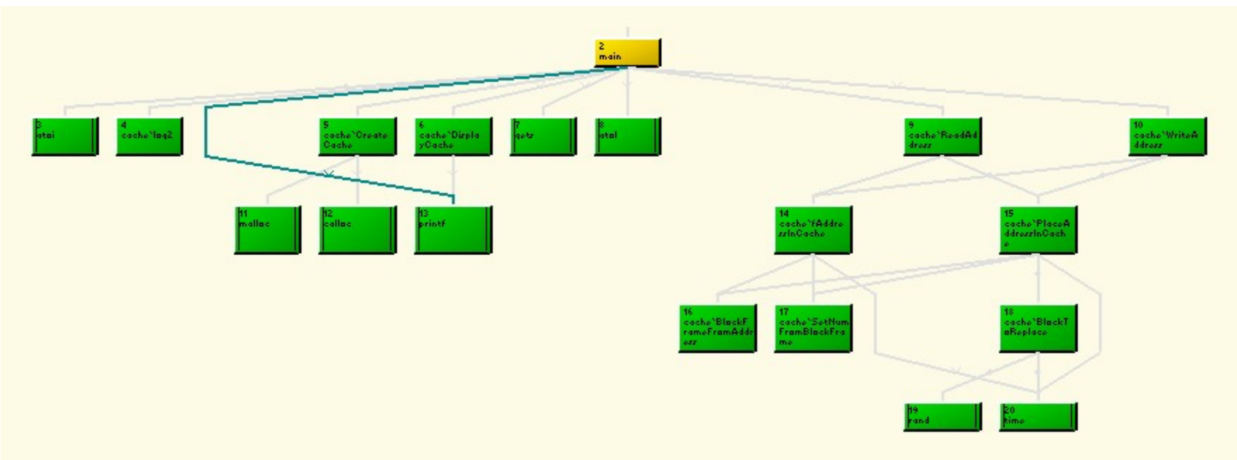
Analyzing a software system for security flaws can be a daunting task, and often begs the question: when is the analysis complete? Often a security analyst will answer this question by determining when they have run out of budget, time or have found bugs. These are not empirical pieces of evidence. One piece of evidence required is to understand how much of the software that is attackable was exercised.

Many experts point out that security requirements resemble those for any other computing task, with one seemingly insignificant difference ... whereas most requirements say "the system will do this," security requirements add the phrase "and nothing more." Not understanding code coverage limits the effectiveness of black-box software testing. If security teams don't exercise code in the application, they can't observe any bugs it might have. Code Coverage measurement should be done on attack surface modules to verify how much of the attackable surface was executed during testing.

Black Box Testing is positive testing. That is, the software security analyst is testing things in the specs or requirements that are already known. White Box Testing is negative testing in that you are testing things that may not be in your specs but in the implementation. Always remember code is what executes not requirements!

Code Coverage monitoring typical of a white box testing approach can verify how much of the source code that is attackable has been exercised using security testing. It can also illuminate how effective security tests actually are after completion. Code Coverage monitoring can give reassurance as to when the security analysis testing is complete and can also indicate what areas of source code penetration tests have hit. The McCabe actual complexity metric indicates how many of the cyclomatic paths were executed during security testing.

The McCabe Battlemap coverage diagram (below) indicates the code coverage of each module, by superimposing test execution information on the structure of the system. This helps you locate attack surface areas of your program that may require additional testing.

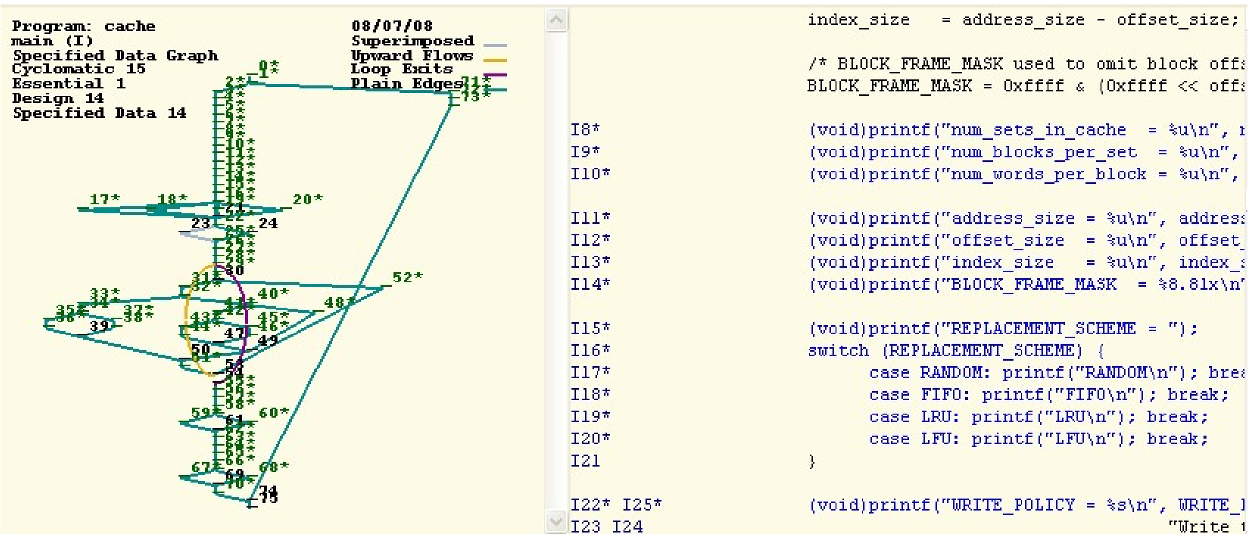


Use Basis Paths & Subtrees for Sneak Path Analysis

Using Cyclomatic basis path testing for software security analysis is analogous to using Sneak Path Analysis. The goal behind sneak path analysis, also called sneak circuit analysis (SCA), is to identify unexpected and unrecognized electrical paths or logic flows in electronics systems, called sneak circuits or paths, that under certain conditions produce undesired results or prevent systems from operating as intended. These paths come about in various ways.

Designers do not always have a complete view of the relationship between functions and components in complex systems, particularly those with many interfaces. System changes may falsely appear to be minor or of only local significance, and users may employ improper operating procedures. Historically, users discovered sneak paths when they observed an unintended effect during system operation. Sneak paths may lie in software, or user actions, or some combination thereof. They are latent conditions, inadvertently and unknowingly designed or engineered into the system, that do not cause system failure until triggered by unusual circumstances.

In Sneak Path Analysis, a topology diagram is built of the various components and interconnections. The system topology diagram is then analyzed to identify ways that sources could combine with targets to cause problems. This is accomplished by examining flow paths through the diagram. An example of the McCabe Cyclomatic Complexity Metric being used for Sneak Path Analysis can be found in *Space Product Assurance Guide: Sneak Analysis Methods and Procedures (ECSS-Q-40-04a) sponsored by the European Cooperation for Space Standardization (ECSS*).



A typical hacker may use features in software which are not obvious security holes, in order to probe the system for useful information to form an attack. Source code complexity can mask potential security weaknesses, or seemingly innocuous software configuration changes may open up security gaps. An example would be a change to the permissions on a system directory.

The implications of such changes may not be appreciated by a system administrator or software engineer, who may not be able to see the whole picture due to the complexity in any typical operating system. A reduction in this complexity may improve security.

Attack Surface Design Complexity

Attack Surface Design Complexity, is a measure of the decision structure which controls the invocation of modules within the design of an attack surface. It is a quantification of the testing effort of all calls in this design, starting with the top module, trickling down through subordinates and exiting through the top.

For example, your attack surface may be in software receiving network packets. What functions are responsible for receiving packets on the network, and how is the resulting data is passed along the internal routines of the software? A suggested process for analyzing control flow of attack surfaces is as follows:

Analyzing an Attack Surface's Control Flow Integrity

Step 1: Identify all modules with vulnerable attack surface

Step 2: Calculate McCabe Design Complexity, Integration Complexity

```
SUBTREES AND ASSOCIATED
INTEGRATION TEST CONDITIONS FOR PROGRAM cache
ROOT MODULE OF PROGRAM: main


SUBTREE #1:
   main > [printf] < main > [printf] < main > [printf] < main

END-TO-END TEST CONDITION LIST FOR SUBTREE #1:
 main 252(1): argc>5 ==> FALSE
```

Step 3: Analyze visual and textual design invocation subtrees

Step 4: Calculate and analyze all cyclomatic, module design, and global data complexity metrics and complexity algorithm graphs for impact analysis, risk, test execution and sneak paths

Step 5: Measure code coverage at point where the packet is received and is traversing memory into the program's logic

Error handling routines in software programs are typically sneak paths. Since error handling routines contribute to control flow, use flow graphs to decipher the programming logic and produce test conditions which will when executed test their logic. The most neglected code paths during the testing process are error handling routines. Error handling may include exception handling, error recovery, and fault tolerance routines.

Functionality tests are normally geared towards validating requirements, which generally do not describe negative (or error) scenarios. Validating the error handling behavior of the system is critical during security testing.

Measuring and Monitoring Code Slices

Measuring and monitoring code execution slices can help uncover your program's internal architecture. By compiling and running an instrumented version of your program, then importing the resulting trace file, you can visualize which parts of your program's code are associated with specific functionality. The slice concept is important to Software Security from several standpoints.

- Visualizing the software
- Tracing data through the system
- Decomposing a complex system
- Extracting business rules
- Tracing requirements
- Finding Sneak Paths

To gain understanding of code coverage before a fuzzing run, it is important to first pass the application a piece of data that is correctly formed. By sending the right packet and measuring the execution slice, the common path that a normal packet takes through the application logic is determined.

Some Intrusion detection systems based on anomaly detection use a set of training data to create a database of valid and legitimate execution patterns that are constantly compared to real execution patterns on a deployed system. This approach assumes that the attack pattern substantially differs from the legitimate pattern.

Detecting Common & Uncommon Code Patterns

Source code metrics can be leveraged to find code patterns, styles and similarities within a codebase. Hackers tend to work in patterns. They learn how to do some things well, and then learn to carry that knowledge and those skills to other areas.

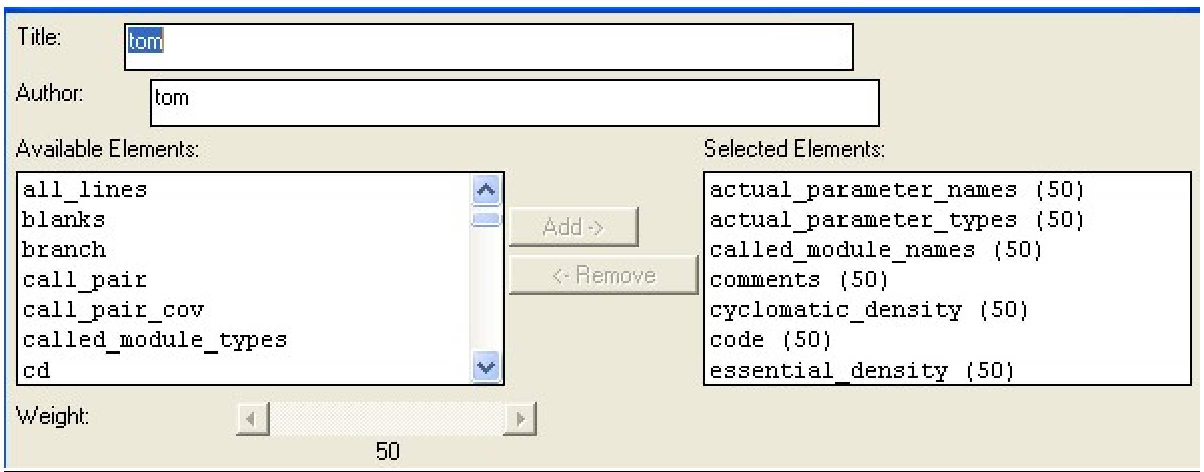
```
Date: 08/07/08
                         Module Comparison Tool Report

Program: cache
Search Criteria: aaa
Search Domain:
          Program Name  Location Date       Title
          ------------  -------- ----       -----
          cache         (pcf)

Maximum similar modules: 10
Minimum similarity score: 75%
Minimum v(G) for search: 3
Total matches found: 20
Module                       Similar Module(s)                      Similarity Score
---------------------------- -------------------------------------- ----------------
cache`BlockFrameFromAddress  (no matches in threshold)
cache`BlockToReplace         main (pcf cache)                       81%
cache`CreateCache            cache`log2 (pcf cache)                 93%
                             cache`WriteAddress (pcf cache)         89%
                             cache`DisplayCache (pcf cache)         84%
cache`DisplayCache           cache`WriteAddress (pcf cache)         94%
                             cache`log2 (pcf cache)                 88%
                             cache`PlaceAddressInCache (pcf cache)  78%
                             cache`fAddressInCache (pcf cache)      76%
cache`PlaceAddressInCache    cache`fAddressInCache (pcf cache)      93%
                             cache`DisplayCache (pcf cache)         78%
cache`ReadAddress            cache`log2 (pcf cache)                 93%
                             cache`WriteAddress (pcf cache)         89%
                             cache`DisplayCache (pcf cache)         84%
cache`SetNumFromBlockFrame   (no matches in threshold)
cache`WriteAddress           cache`DisplayCache (pcf cache)         94%
                             cache`log2 (pcf cache)                 93%
cache`fAddressInCache        cache`PlaceAddressInCache (pcf cache)  93%
                             cache`DisplayCache (pcf cache)         76%
cache`log2                   cache`WriteAddress (pcf cache)         93%
                             cache`DisplayCache (pcf cache)         88%
main                         cache`BlockToReplace (pcf cache)       81%
```

Where Else is That Within My Codebase? What Else is Similar?

Module comparison tools can be used to locate redundant code. A predefined search criteria can be used or establish your own search criteria for finding similar modules by selecting any of several hundred source code metrics. After you select the search criteria, select the modules you want to match, and specify which programs or repositories you want to search, the module comparison tool locates the modules that are similar to the ones you want to match based on the search criteria you selected. Often times, the code associated with individual programmers can be identified by the code constructs they use. Source code containing known security flaws can be analyzed and used as a baseline to compare against the codebase under security review.



Conclusion

Use software complexity metrics, measure control flow integrity and do sneak path analysis for better security analysis. Many of the issues surrounding Security Analysis are intertwined with fundamental software engineering principles. Metrics such as the Relative Attack Surface Quotient (RASQ) from Microsoft should be used in conjunction with traditional metrics that enable us to understand software and test it. Complexity, Object-Oriented Metrics and other metrics that help us understand the characteristics of our codebase are certainly conducive to good software security analysis.

Software testing and code coverage metrics can also provide insight into the security analysis of a valued codebase. Basis cyclomatic test path and subtree analysis lends itself well in the area of software Sneak Path Analysis. White box security testing following the methodology as presented in NIST Special Pub. 500-235 Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric is a sound way to verify control flow integrity. Control Flow Integrity and Sneak Path Analysis should be a part of Security Analysis discussions.

[1] Testimony of Bruce Schneier, Founder and CTO Counterpane Internet Security, Inc, Subcommittee on Cybersecurity, Science & Research and Development Committee of Homeland Security, U.S. House of Representatives , Jun 25, 2003