Software Security Analysis:
Control Flow Security Analysis with McCabe IQ

Applying a Path-based Method to Vulnerability Assessment of the Microsoft SDL Banned Function Calls

Introduction

When considering software security analysis, the primary concern is to ensure that the system is resistant to malicious misuse. However, security vulnerabilities are also closely related to structural quality and implementation flaws. As software becomes more complex, security flaws are more easily introduced and more difficult to eliminate.

Comprehensive security analysis requires insight into the structure of the software code, to manage complexity and track possible execution flows, ensuring that all paths of execution are valid and secure. Therefore, in addition to identifying potentially vulnerable areas, security analysis tools must also be cognizant of related control flow paths surrounding them.

McCabe IQ is a source code analysis tool with a strong focus on function call relationships and control flow paths. While many tools commonly detect potentially insecure code patterns, function calls, or expressions, discovery of the surrounding context is often left for the analyst to manually infer. McCabe IQ mitigates this arduous task, taking into account the complexity and connectedness of components when analyzing vulnerability.

One of the industry leading processes for secure software engineering is the Security Development Lifecycle (SDL) developed by Microsoft. The SDL defines a workflow that incorporates security-related activities throughout software development. McCabe IQ's capabilities are best suited to the activities defined in the design, implementation, and verification phases of the SDL. Some of these activities include attack surface analysis, static source code analysis, and testing. McCabe IQ is designed to facilitate these efforts.

This application note discusses the example of performing vulnerability assessment in relation to the use of certain exploitable functions in the C standard library. As part of the recommendations for the implementation phase, the Microsoft SDL identifies a set of functions that, from real-world experience, have been linked to many security bugs because of buffer overruns and invalid pointer access. SDL practices suggest banning the use of these functions in favor of newer implementations that incorporate better bounds checking and are easier to secure.

Searching source code for banned function calls will readily identify the vulnerable points, but the exploitability of a given vulnerability is determined by whether it is reachable along an execution path from parts of the system accessible to an attacker. Exploitable vulnerabilities call for special attention to design remediation and adequate testing. The following sections describe activities that apply such practices using McCabe IQ.

Analyzing Use of Banned Functions with Attack Maps

Attack maps are control flow diagrams that identify a set of interconnected routines in a system, that potentially participate in a malicious attack. An attack map is intended to show how call relationships and a flow of execution can connect externally triggered attacks to critical areas of the system.

The main window of the McCabe IQ application consists of a structure chart called the battlemap. This chart shows functions as boxes connected by lines indicating function call relationships. Attack maps allow you to filter the battlemap chart in a way that focuses attention on routines that lie along attackable execution flows.

This section covers the following topics:

- Definitions
- Mapping the Banned Functions with McCabe IQ
- Further Analysis Activities
- Applying Code Coverage to Attackable Space

Definitions

An attack map connects two significant areas of interest: Attack Surface and Attack Target. The attack surface is generally known as the subset of input space with which a malicious user can exploit the system by giving it malformed data to trigger deviant behavior. One of the heuristics of securing software is reducing the attack surface. That is, minimize the number of external interfaces that influence system behavior. In a general sense, an attack surface encompasses code, interfaces, services, and protocols. However, within the scope of source code analysis, the relevant attack surface consists of the areas of code where the system obtains external input. For example, the analyst might focus on input routines that accept data or read configuration files, environment variables, or registry entries that affect application behavior. It is important to identify these entry points and review them to assess their correctness and robustness. It is from this space where a malicious attack will originate.

The other area of equal interest is called the attack target, which is defined as the areas of the system that can cause adverse critical impact if exploited. The banned functions fit this category. Misuse of these APIs can cause significant consequences ranging from wasting system resources to program crashes and security breaches. Given the attack surface and attack target, McCabe IQ can analyze function call relationships and direct attention to the routines that connect these two areas of concern.

Mapping the Banned Functions with McCabe IQ

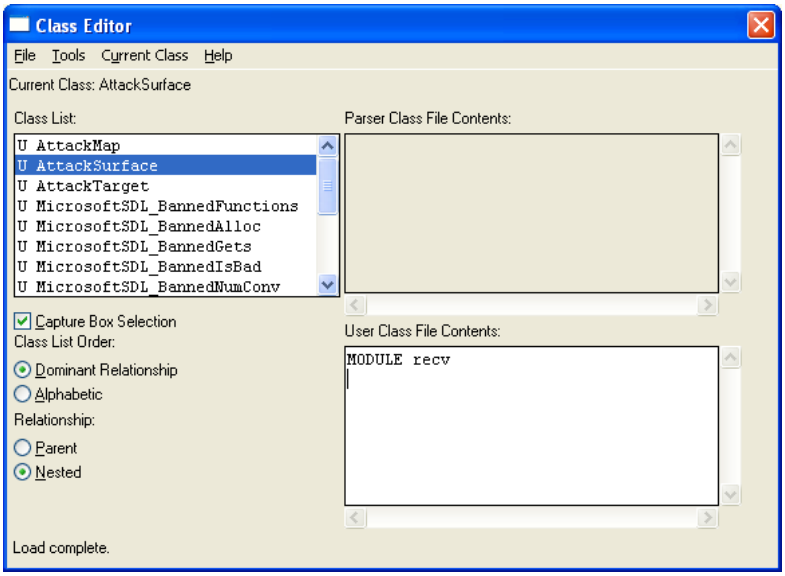
Two McCabe IQ features are integral to mapping the banned functions. They are as follows:

- Using the Class Editor to Identify Attack Surface and Attack Target
- Using the Exclude Feature to Narrow the Scope of Analysis

Using the Class Editor to Identify Attack Surface and Attack Target

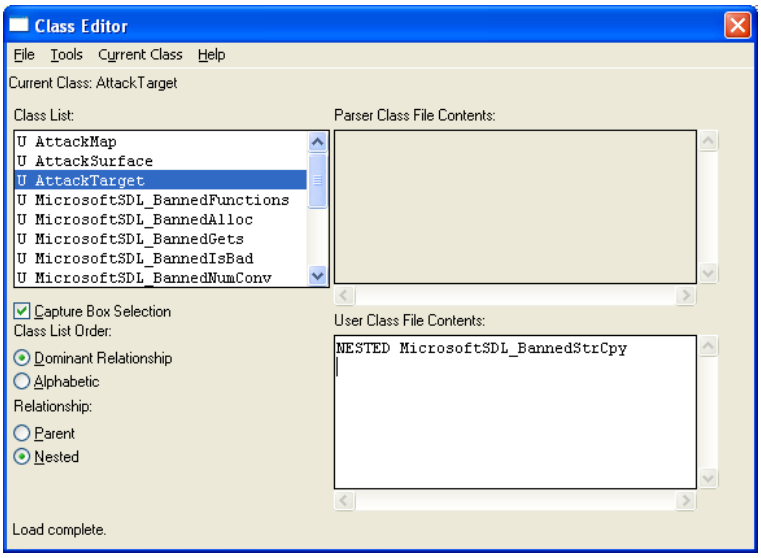
To create an attack map with McCabe IQ, the user must first identify functions in the attack surface and attack target groups. This is done by using the McCabe IQ Class Editor, available from the View menu of the main application window. For purposes of attack maps, classes are simply used as a way of grouping functions together. Analyzing calls to banned functions in the standard C library is particularly easy because the list of these functions is readily available, and is easily preconfigured in the McCabe IQ Class Editor.

To identify functions in the attack surface, create a class called AttackSurface, and add to its contents the routines from the input space that you wish to trace (see the screenshot below). For example, in a network application, the recv() function, which receives data from a socket, may be of interest. With the AttackSurface class highlighted from the class list, you can manually type in `MODULE` *`modulename`* in the User Class File Contents box, or simply clicking a module box in the battlemap will add it to the current class. You can add as many functions as you wish to the AttackSurface class, although the more functions you add, the more inclusive (larger) the map will be. You can also use the `NESTED` *`classname`* specification to add a group of functions as part of the AttackSurface class. The AttackTarget class in the following paragraph of this example uses this technique.



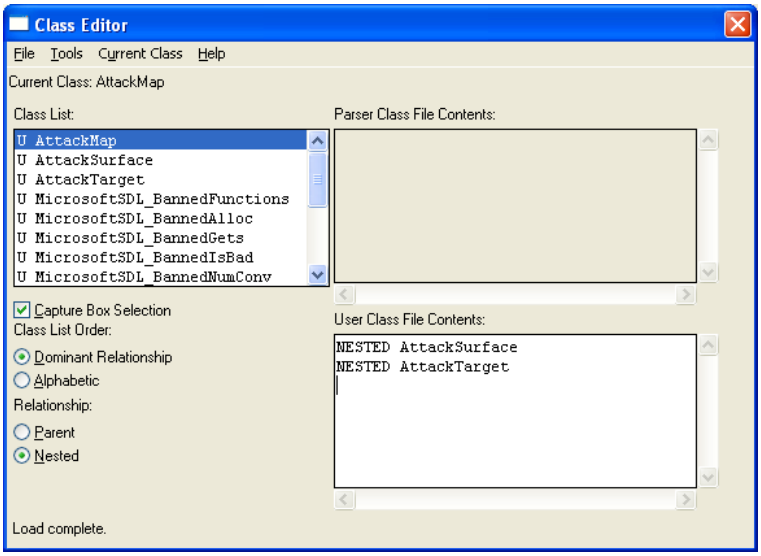
To configure a set of banned functions as the attack target, create a class called AttackTarget and add to its contents the subset of banned functions you are interested in tracing (see the screenshot following). In this example, the AttackTarget is configured to consist of the banned variants of strcpy(). Notice that the syntax in the User Class File Contents box says `NESTED MicrosoftSDL_BannedStrCpy`. This is because the prepopulated class list includes a class named `MicrosoftSDL_BannedStrCpy`, whose contents consist of the banned strcpy() function names.

As with the attack surface, you can also specify individual functions using the `MODULE` *`modulename`* specification. Again, you can add as many target functions as you wish to the AttackTarget class. The screenshot below illustrates how you can specify `NESTED` *`classname`* to nest group definitions and create aggregate hierarchies according to your needs. This allows flexibility for easily configuring the function calls you wish to trace.



Finally, you must define a class called AttackMap that nests the AttackSurface and AttackTarget classes (see the following screenshot). The AttackMap class comprises both the AttackSurface and AttackTarget groups. Note that you are not required to use the specific class names AttackMap, AttackSurface, and AttackTarget. What is important is that you have identified and created two groups of functions to

represent attack surface and attack target, and a third group (attack map) to tie the two together using the NESTED specification. For example, you may have several sets of three (surface, target, map) groups that you wish to maintain, in order to map different attack models.

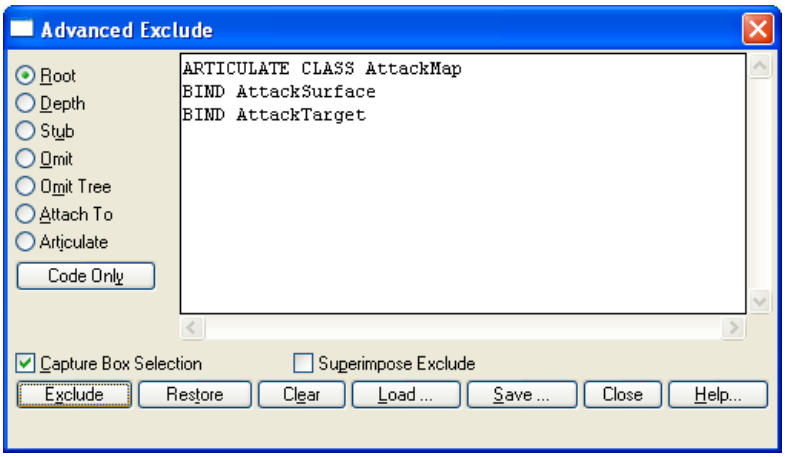


Using the Exclude Feature to Narrow the Scope of Analysis

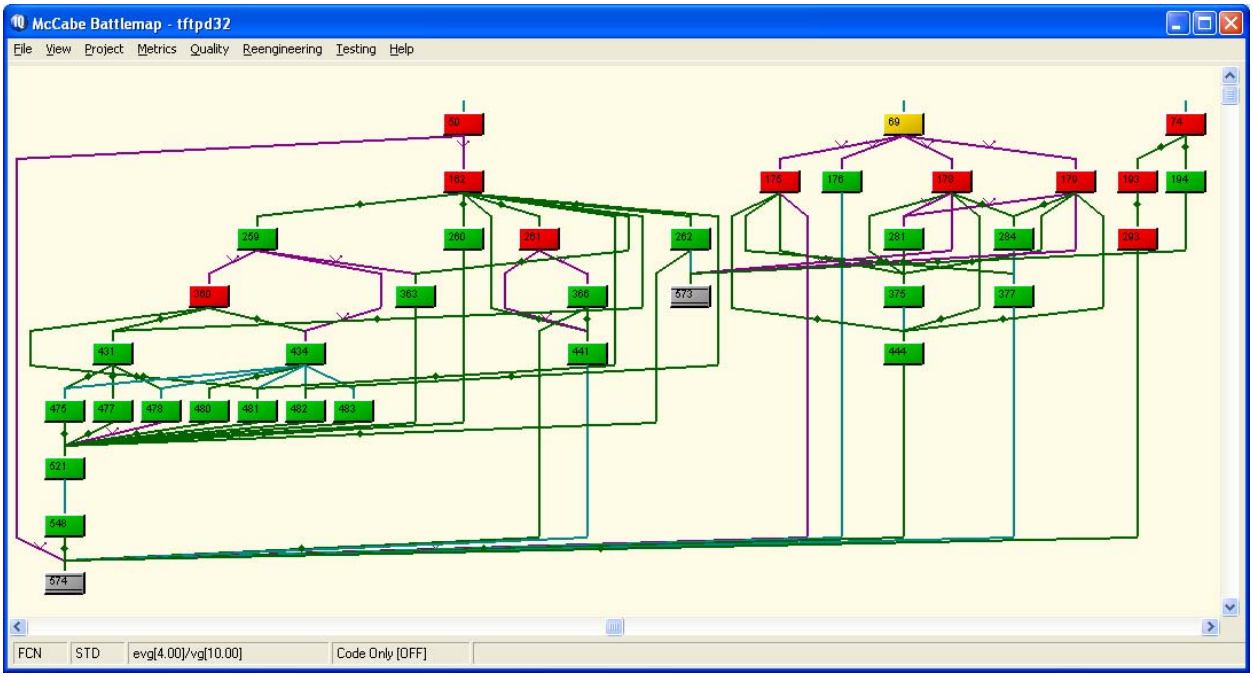
The second step in attack map modeling is to use McCabe IQ's Advanced Exclude feature (available from the View menu of the main application window) to filter out functions that do not participate in the call relationships between attack surface and attack target. The Advanced Exclude dialog offers a selection of exclusion commands that can be applied to the battlemap chart. From the class definitions previously created, the fundamental command relevant to producing an attack map is `ARTICULATE CLASS AttackMap`. Since the AttackMap class nests both the AttackSurface and AttackTarget groups, this command will articulate on the functions related to the two sets.

When you apply the commands by clicking on the Exclude button, the resulting chart will consist only of functions in the attack surface group, functions in the attack target group, and functions that have call relationships with those groups. All other functions will be filtered from view.

If you have defined multiple functions in the attack surface and/or attack target it may be simpler to show them as a group in the chart. You can do this by adding the commands `BIND AttackSurface` and `BIND AttackTarget` to the exclude dialog. This command collapses functions of a specified class into one box on the chart.



Following is an example of how the chart may appear after applying the three exclude commands. As a reminder, the default red, yellow, and green color coding of the boxes represent high, medium, and low complexity functions, respectively. The two grey boxes represent the AttackSurface and AttackTarget groups.

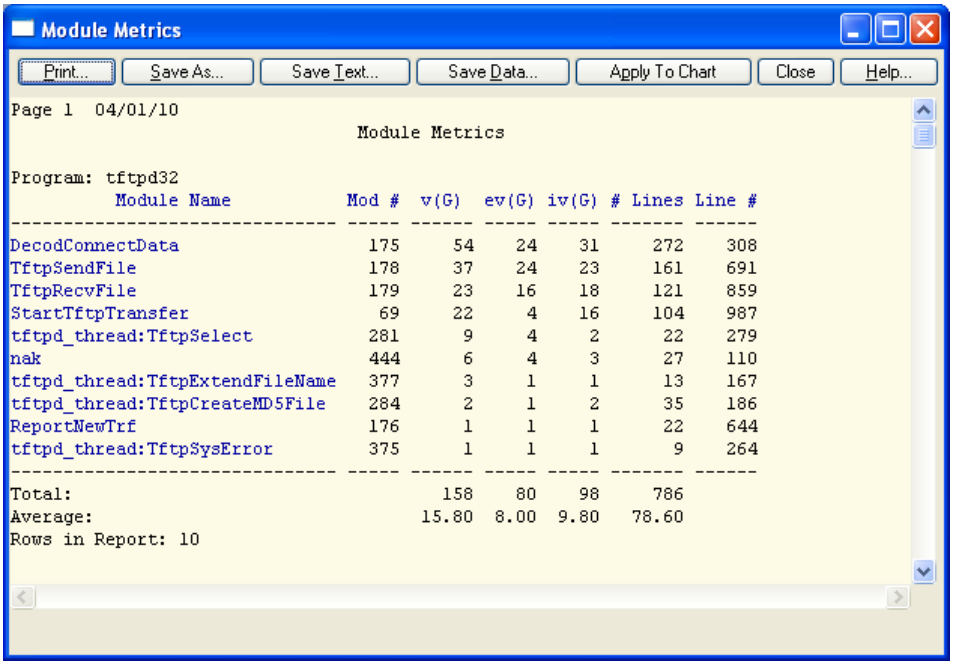


Thus, after excluding based on your specified classes, the view shows only a subset of the functions that make up the application. Specifically, it shows only the functions that participate in connecting AttackSurface and AttackTarget. In this example, it shows the call relationships connecting recv() and banned strcpy().

This filtering helps you focus the effort and scope of analysis on functions and paths that are potentially exercised in the execution flow of an attack. The control flows within and between these functions deserve special attention, and must be verified and tested adequately.

When the battlemap chart shows a filtered subset of functions in the application, the other features of McCabe IQ abide by the same filtering. For example, generating a basic module metrics report will show metrics for only the same functions that are visible on the chart, instead of reporting on all the functions in

the application.



This keeps the user focused on analyzing the prioritized subset. The usual McCabe analysis features can be applied to further scrutinize only these critical routines. You can analyze module flowgraphs and source code listings for the functions of concern. Various reports and metrics provided by the tool help determine which functions have a high risk, and gauge the testing effort based on function complexity.

Since these functions lie along the attackable execution flow, software developers might consider remediation of risky functions through refactoring. For example, from the report above, we see that four of the functions in the attack map have a cyclomatic complexity metric [shown under the v(G) column] over 20. Complete basis path testing for these functions would require numerous test cases. A suggestion to consider would be to refactor and break down the complex functions into smaller functions. Hypothetically, only a portion of those smaller functions would remain in the attack path of execution, effectively reducing the complexity of the attackable space.

Less complexity also facilitates testing, requiring fewer test cases to achieve complete basis path coverage.

Further Analysis Activities

After applying attack maps to prioritize the functions under security review, there are a number of other detailed analysis activities that McCabe IQ facilitates. Although attack maps can focus the scope of analysis to a subset of functions, it is necessary to investigate those functions in detail, to assure structural quality and validate all paths of execution. Following are some suggested activities:

- Articulating Other Calls to the Attack Targets
- Examining Individual Root Modules and Associated Subtrees
- Investigating Control Flow Paths in a Function
- Using the Data Dictionary to Investigate Specific Control Flow Paths

Articulating Other Calls to the Attack Targets

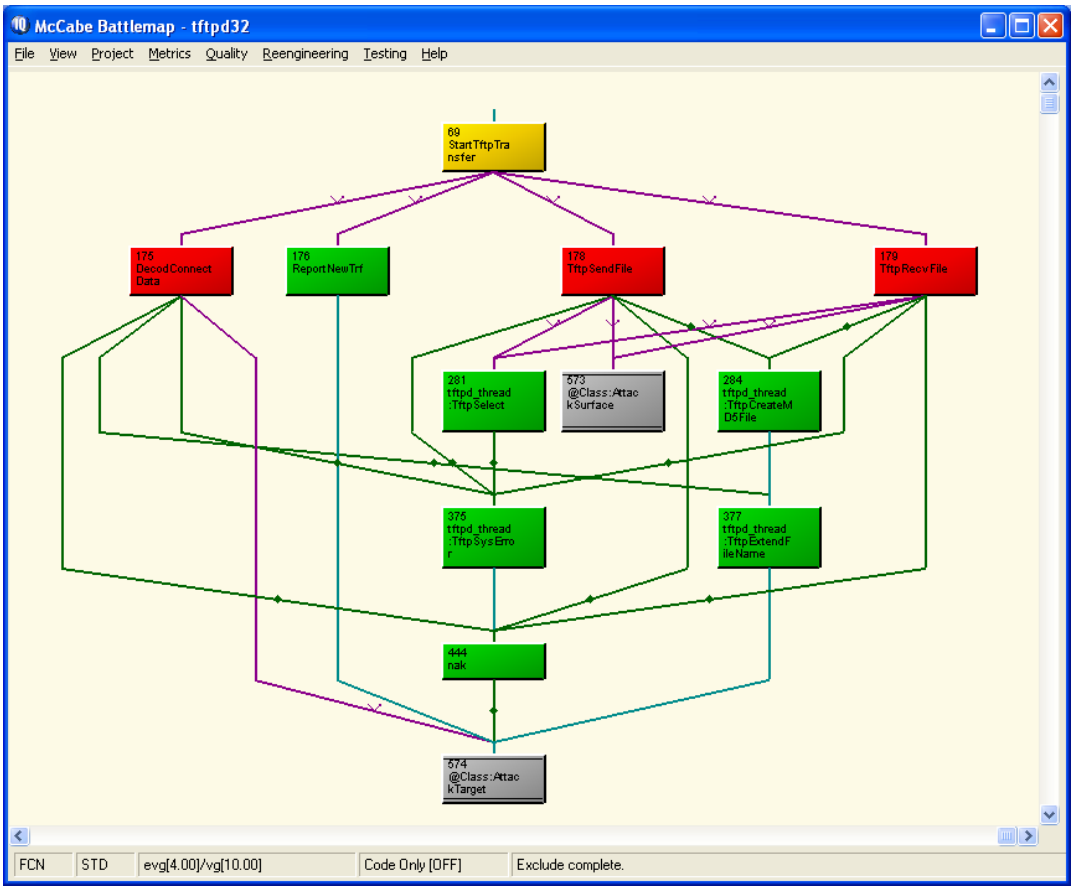
While attack maps focus on call relationships that are reachable from the attack surface, it is worth making note of other possible ways to invoke the vulnerable attack targets. The Advanced Exclude

feature can articulate on a single function or a single class (group of functions), to show the call tree into the specified item. This often leads to finding subtle defects that, although not directly reachable from an attack surface, may still cause undesired consequences to application behavior.

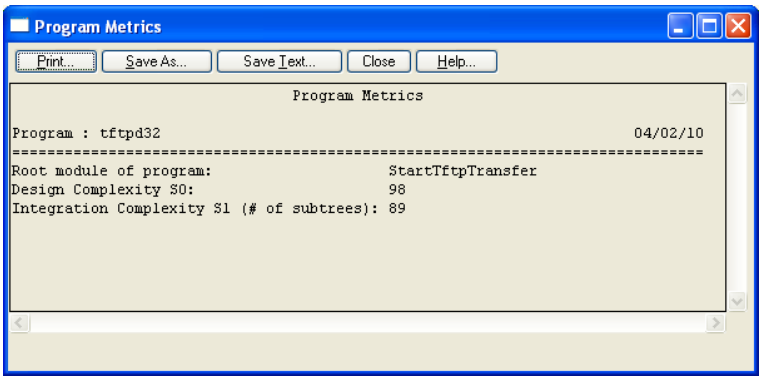
Examining Individual Root Modules and Associated Subtrees

Having focused the scope of analysis to a priority subset of functions, it is important to investigate each root level function and examine the integration paths that reach the attack target. The McCabe IQ battlemap is a structure chart showing a hierarchical call tree. Root level modules are functions to which the tool found no direct calls. Some applications have a single root module representing the main entry point of the program. However, other systems, like event driven applications, may have multiple entry points. We recommend analyzing each call tree, to be cognizant of the integration paths that include the attack target.

To focus analysis on an individual call tree, add the `ROOT` command in the Advanced Exclude dialog. This will show only the functions in a call tree rooted at the specified function. For example, following is the chart after adding the command `ROOT StartTftpTransfer` and reapplying the exclusion:

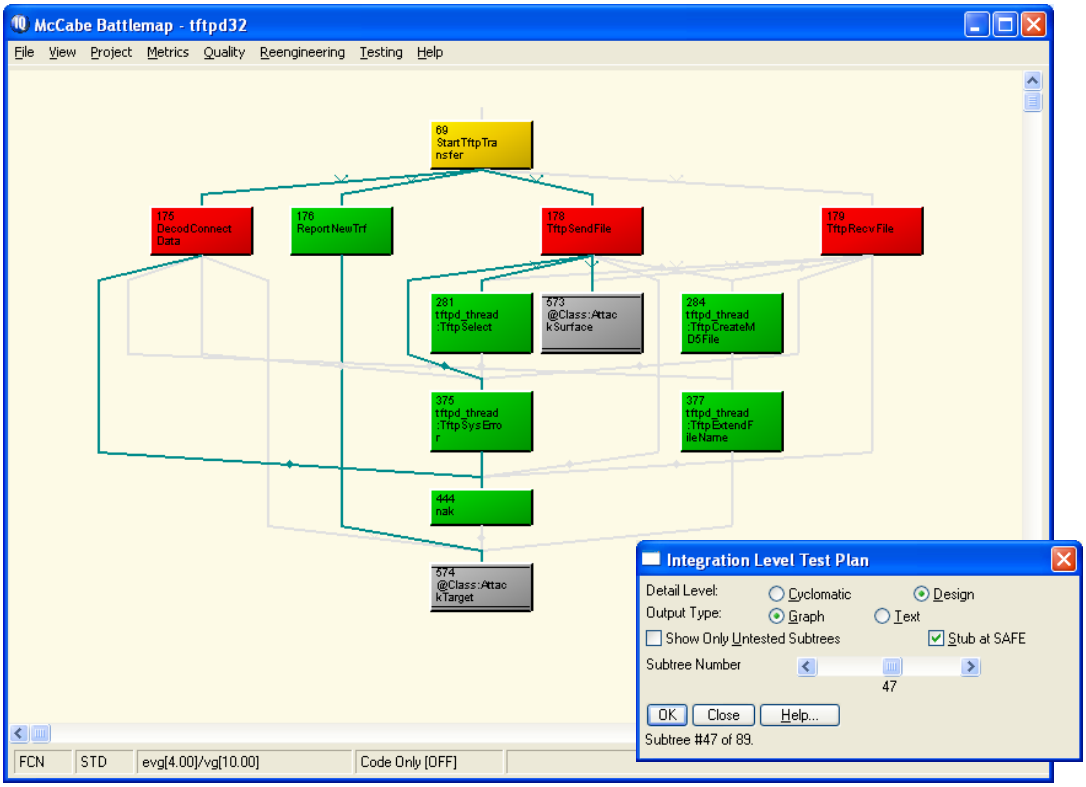


As noted before, other battlemap features abide by the currently displayed functions. Generating a program metrics report would show the program design complexity and integration complexity for the subset of modules with the single current root.



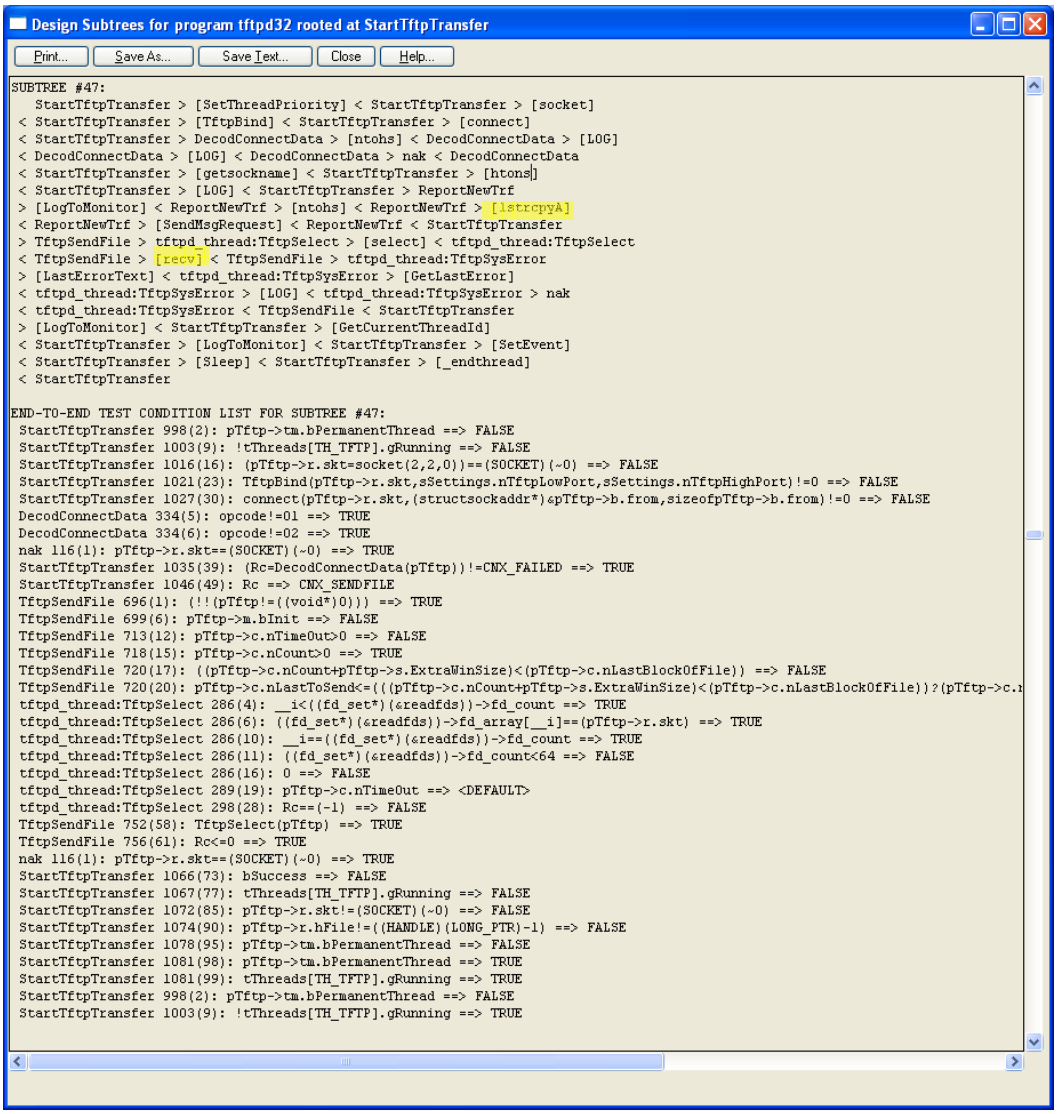
Recall from basic McCabe IQ concepts that the program integration complexity represents the number of linearly independent paths through an entire program's design. In this example report, the integration complexity of the subset of functions rooted at StartTftpTransfer is 89. This means there are 89 unique paths to fully exercise a set of all linearly independent paths through the functions shown in the chart. McCabe IQ can highlight the control flow paths and show the sequence of decision outcomes needed to exercise them.

To see the linearly independent integration paths through the functions in the current chart, use the Integration Level Test Plan feature (available under Testing->Test Plan menu of the main application window), and generate design subtrees. If you set the output type to Graph, you can use the Subtree Number scroll bar to step through the various subtrees in the chart. As you do so, the chart will highlight certain lines connecting the boxes. This represents the calls exercised for the selected call tree sequence. In the context of attack maps, special interest is called for on specific integration subtrees that highlight the attack target, especially those that highlight calls to both the attack target and attack surface.



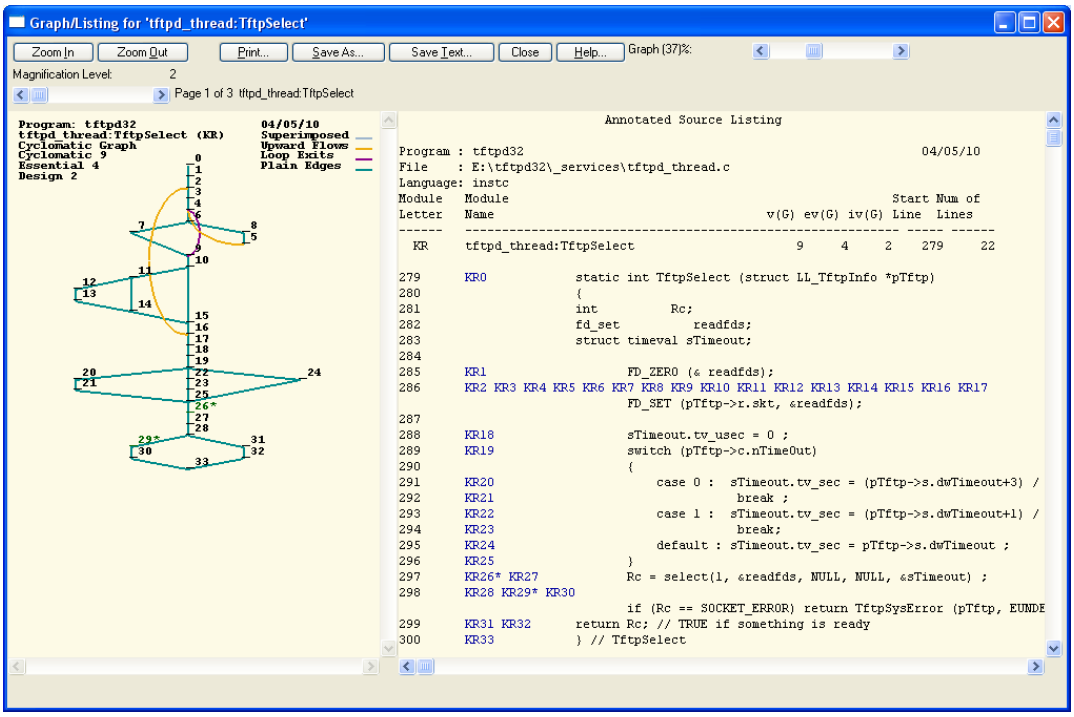
If you set the output type to Text when generating the design subtrees, McCabe IQ will open a window

with a text report showing the details of the function call flow, listing the calling and called modules plus the test conditions needed to exercise the subtree. For example, subtree 47 highlighted in the chart above shows details like the following in the textual report (highlights added to show where recv() and strcpy() participate in the call tree). Details on the notation for calls to and returns from other functions are explained in the McCabe IQ user manuals.

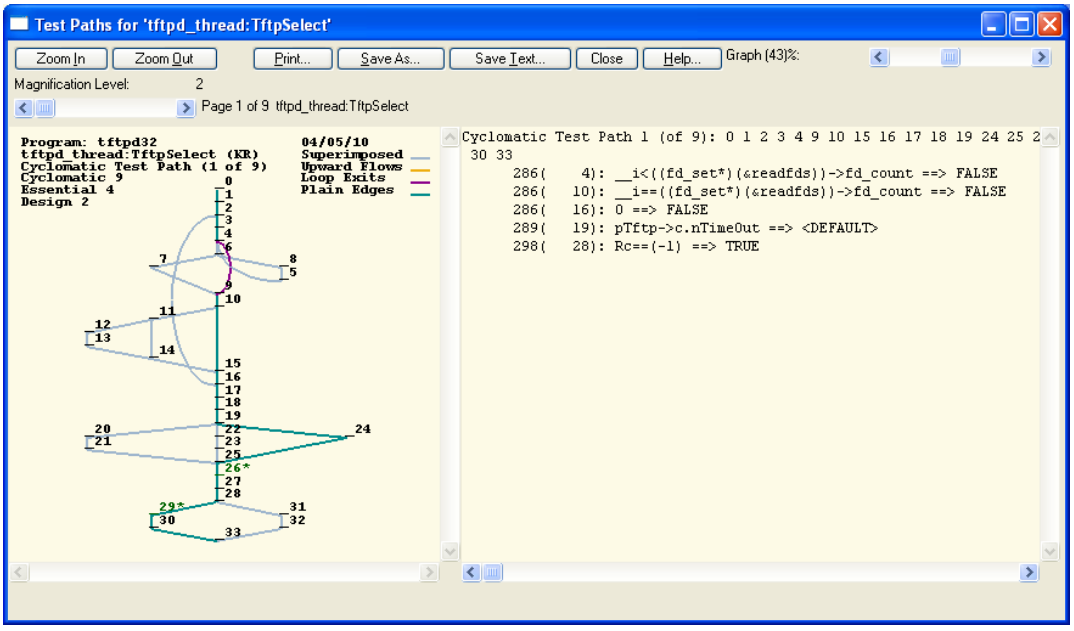


Investigating Control Flow Paths in a Function

You can right-click on a box on the battlemap chart, or on a function name in a report to bring up a context menu that allows you to drill down into the details of that function. Two items of particular interest are flowgraphs and test paths. If you select “Graph/Listing…”, McCabe IQ will open a dual pane window showing the implementation details for the selected routine. On the left pane is a graphical representation of a routine’s control flow. The right pane shows the source code for that routine. Node numbers on the graph are also displayed on the right (shown in blue in the screenshot below), so you can see which lines of code correspond to the nodes on the graph. You can right-click on a node number in the source code and highlight the corresponding node on the flowgraph. Conversely, you can also right-click a node number on the flowgraph and highlight the corresponding line of code in the source code pane. The screenshot below shows an example flowgraph and source listing for a given function.



Another key item from the context menu is the “Test Paths…” item. Selecting this option opens a similar window, but instead of showing the source code on the right, it shows a set of conditions for a specific path through the function. Furthermore, the flowgraph on the left highlights the path that the conditions will exercise. The screenshot below shows the test paths window for the same function shown in the previous screenshot. Note that it is showing the first of nine linearly independent paths. The scroll bar under the Zoom In and Zoom Out buttons allows you to step through all nine paths identified by McCabe IQ.

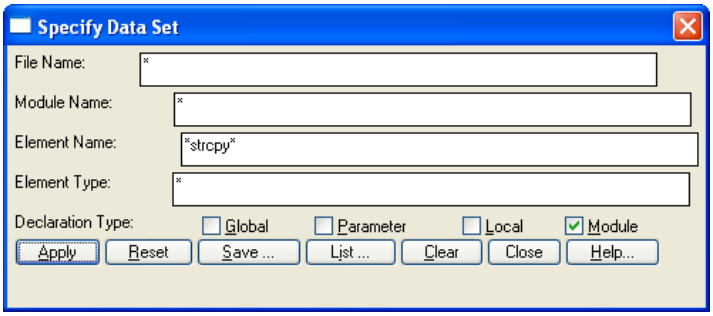


With the flowgraph, annotated source code listing, and test path details, you can carry out a thorough analysis of each function in the attack map, to ensure that the control flow paths shown are valid, consistent with requirements, and secure.

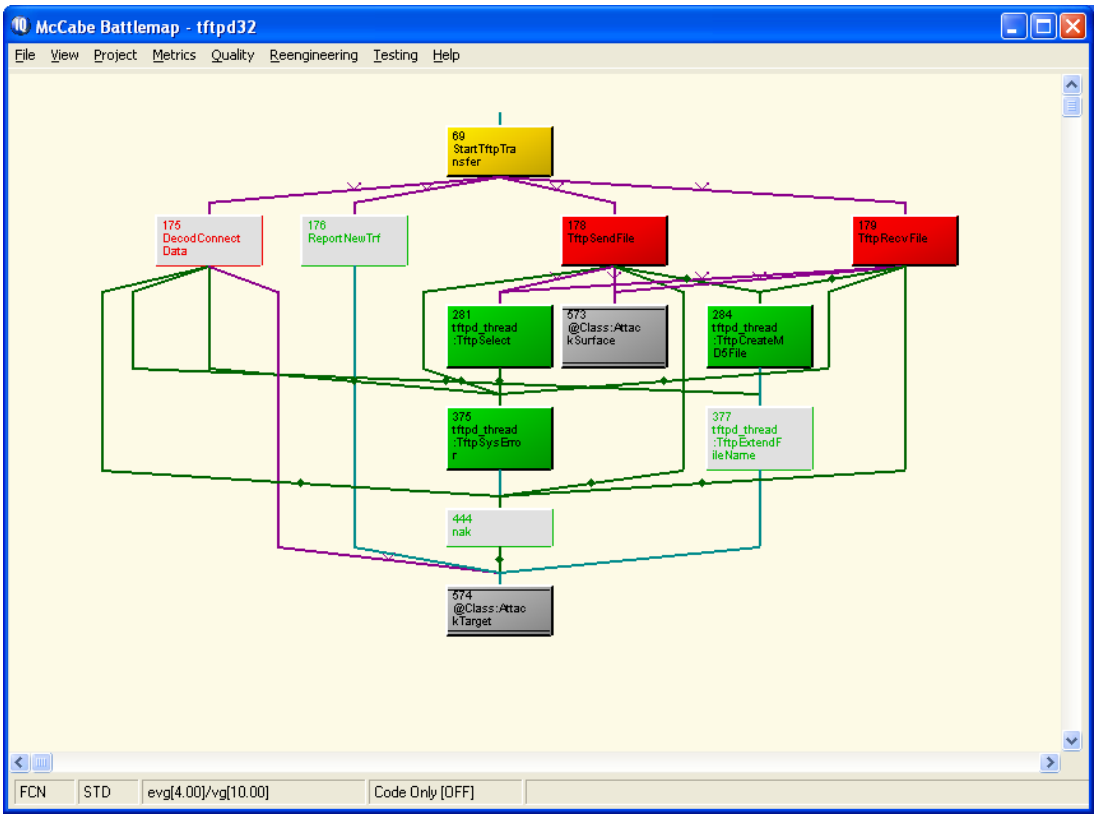
Using the Data Dictionary to Investigate Specific Control Flow Paths

The Data Dictionary feature of McCabe IQ (available from the Reengineering->Data menu of the main application window) is another technique to do more detailed investigation of where in the application code banned functions are being used. The data dictionary can search data elements and function calls, and identify the paths within a function that entail those calls.

Continuing with the above example, you can configure a data set that consists of calls to the banned strcpy() functions. From the Data Dictionary window, click on the “Data Set…” button, and search for module elements named “*strcpy*”. In the Data Set dialog, click the “Apply” button. This will find (from the set of functions currently visible in the chart) all instances where banned strcpy() variants are invoked, and apply it as the current specified data set.

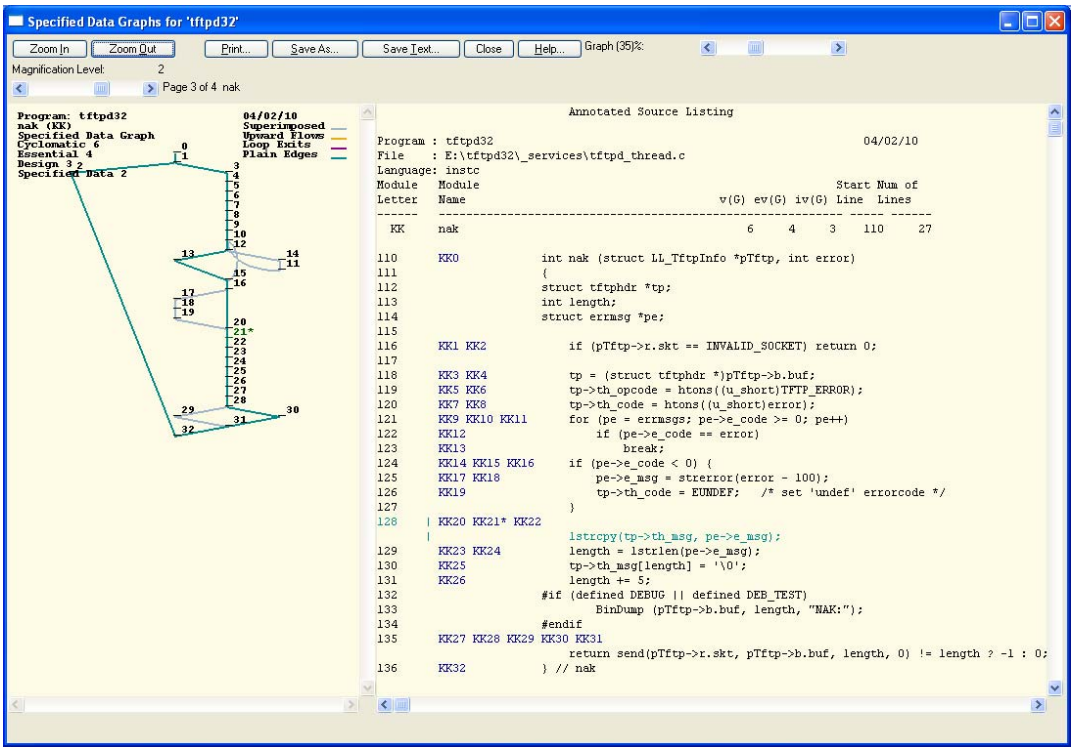


Once this data set is applied, you can highlight which functions in the battlemap chart contain the data set elements. To do this, select “Highlight->Data Set” from the menu on the Data Dictionary window. The battlemap chart will show the functions that directly invoke the matching elements. Following is what the chart may look like. Notice that all the highlighted functions call directly into the attack target group, which, in this example, represents the banned strcpy() functions.

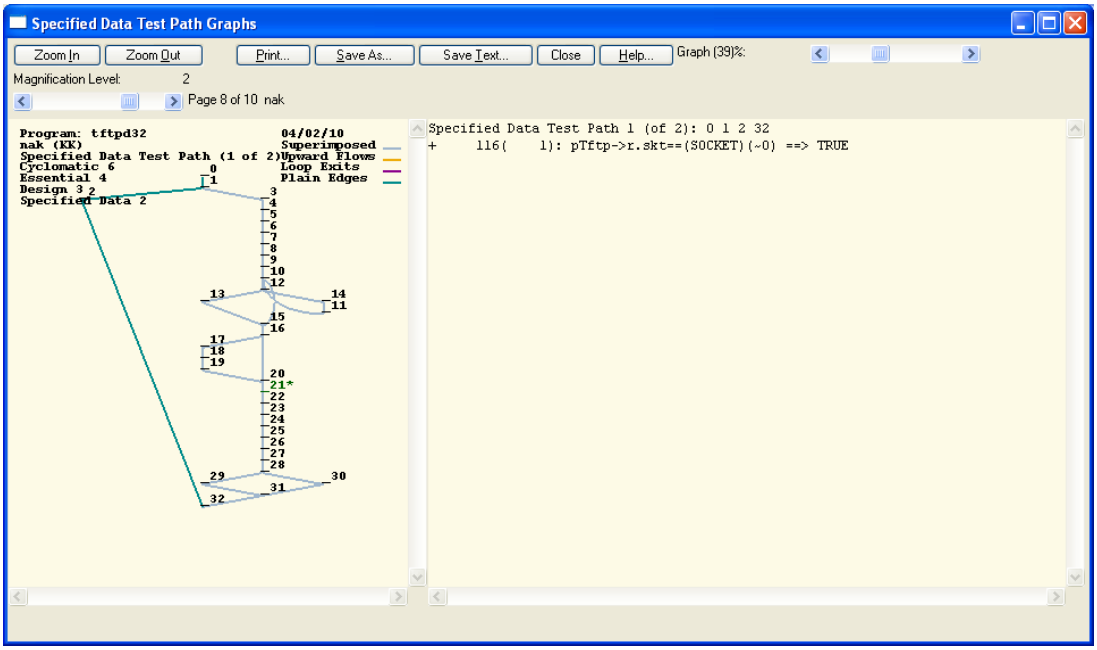


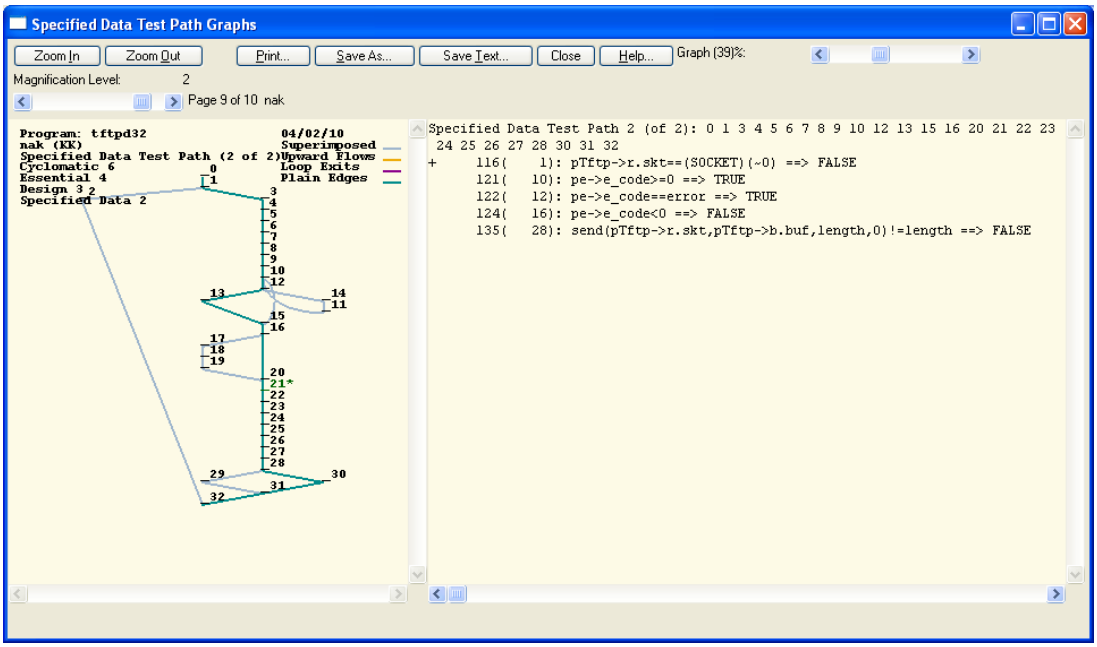
You can also look at individual control flow graphs for the currently specified data set by selecting “Reports->Specified->Graph/Listing…” from the menu on the Data Dictionary window. This brings up a window with a flowgraph and annotated source listing of each of the functions in the current specified data set. This is similar to the flowgraph and source code window described earlier, except that this flowgraph will highlight path flows through the function on which the specified elements lie. Furthermore, the source listing on the right pane will show the source code for the function, highlighting the lines that contain the specified data set elements.

In the example below, the specified data graph/listing window shows function nak() calling lstrcpy(), highlighted on line 128. The control flow graph is shown on the left pane. This function has a cyclomatic complexity metric of 6, with a specified data complexity of 2. This means that there are 6 linearly independent paths through this function. Of those 6 paths, 2 independent paths have relationships to a call to strcpy(). At minimum, there should be tests that exercise those 2 paths.



To find the conditions necessary to exercise the 2 specified data paths, select “Reports->Specified->Test Paths…” from the menu on the Data Dictionary window. This will open a Specified Data Test Paths window that allows you to step through each of the specified data test paths for each of the modules in the currently specified data set. The left pane shows the control flow graph of the function, highlighting the specified data test path. The right pane shows the conditions needed to exercise that path. The following two screenshots show the 2 specified data paths and conditions from the above example.





Applying Code Coverage to Attackable Space

After identifying and examining details of the functions in the attack map, it is important to know how thoroughly these critical functions are tested. One of the major benefits of McCabe IQ is that it allows you to perform both structural complexity analysis and coverage analysis with one integrated tool. McCabe IQ includes features to instrument your code, and report on the paths that have been exercised as a result of running your tests. The following procedures are described:

- Using McCabe IQ Source Code Instrumentation
- Reporting on Path coverage
- Exercising Remaining Untested Paths

Using McCabe IQ Source Code Instrumentation

McCabe IQ can export an instrumented copy of the source code it has analyzed. Instrumented code is a copy of the original source code, augmented with counters to track coverage on code as it is executed. The user can then build the instrumented copy of the source code, to produce an instrumented application that collects coverage data. If you are using the Microsoft Visual Studio IDE, McCabe IQ provides a Visual Studio Add-in that facilitates this process.

When an instrumented version of the application or component (.exe, .dll, etc.) is built, you can deploy it in place of the normal binary and run your security tests. The application should run in a logically equivalent behavior to a normal build, except that the instrumented component information as to which parts of the code are being exercised. This data is collected in a trace output file that will be imported back into the McCabe IQ application, to see the results of execution.

Reporting on Path Coverage

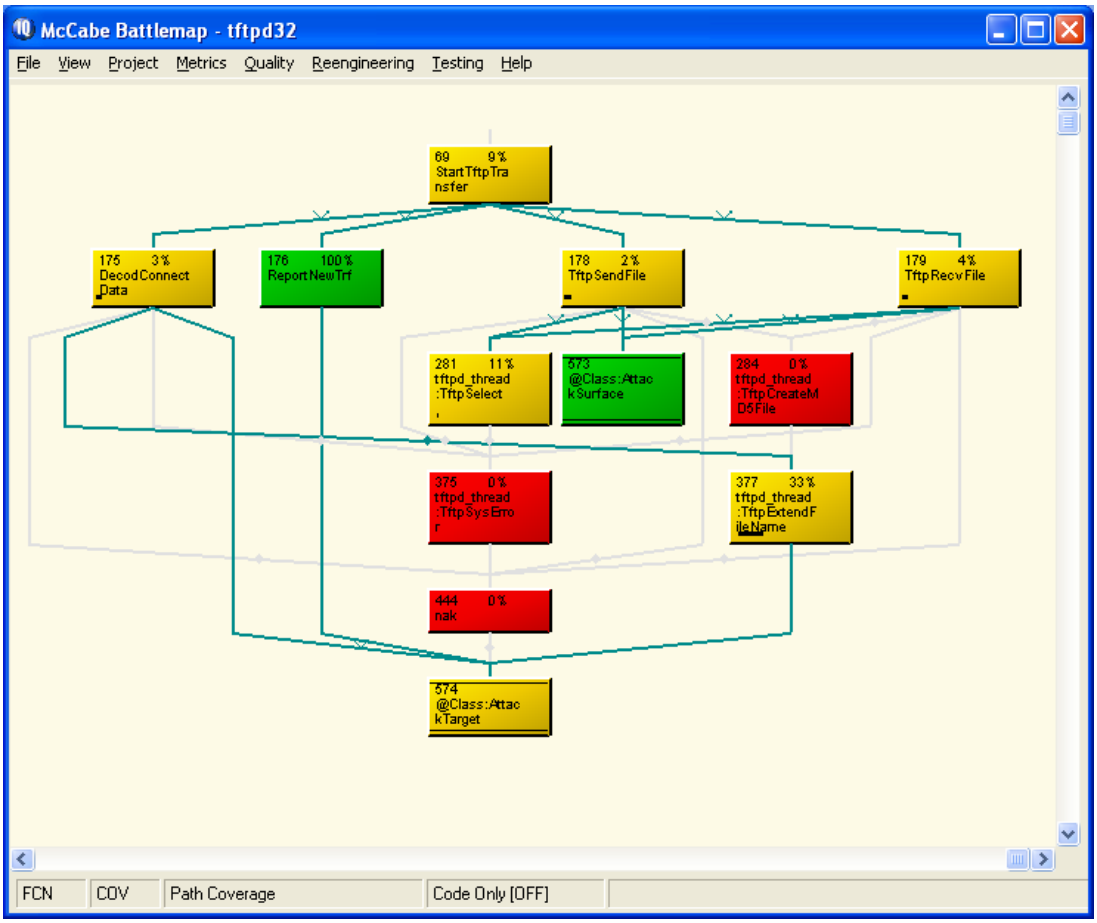
To incorporate coverage information into the McCabe IQ analysis, select “Testing->Testing Data->Import…” from the McCabe IQ Battlemap menu. Then import the trace file that corresponds to the currently loaded program. When the trace data has been loaded, select “View->Switch To Coverage Mode”, to see the coverage information on the chart.

In coverage mode, the red, yellow, and green coloring scheme represents untested, partially tested, and fully tested functions, respectively, based on the currently selected coverage technique. The boxes can

also show the percentage of coverage, and the lines showing call relationships are also highlighted according to coverage.

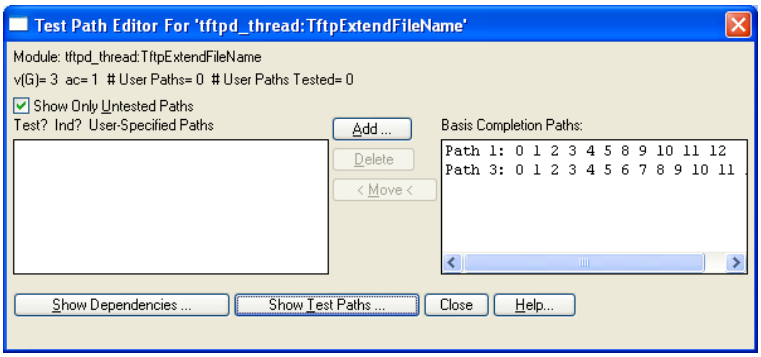
McCabe IQ supports various coverage measurement practices including statement, branch, boolean, and basis path coverage techniques. Basis path coverage provides the most thorough level of testing, and is highly recommended for critical security applications, especially for the most critical code components of these applications. Basis path coverage is based on linearly independent paths, which is the hallmark feature of McCabe IQ. By default, the coverage indicators (colors and percentage numbers) used in the chart are calculated based on basis path coverage.

If you have the chart pared down to show only the functions in the attack map, coverage mode will allow you to investigate in detail, which functions and call trees have been exercised. This effectively measures how well the tests have exercised the code in the attackable space. The chart below shows an example with partial coverage of the attack tree rooted at StartTftpTransfer. Many functions are only partially tested, and some lines representing call relationships have not been exercised. Since these functions lie along an attackable call tree, your tests should be designed to maximize coverage in these areas. Ideally you would want to see green boxes with all call lines highlighted, for the attack map in coverage mode.

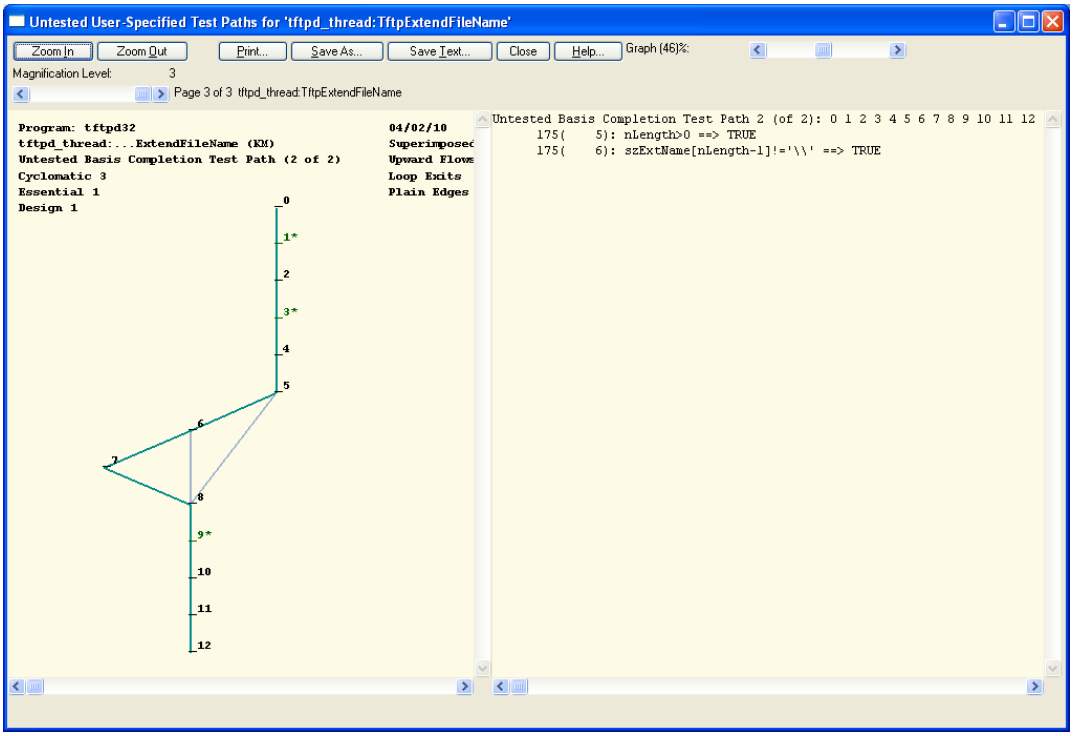


Exercising Remaining Untested Paths

For a given function, McCabe IQ can analyze the current coverage levels and suggest specific control flow paths to exercise in order to achieve complete basis path coverage. Using the Test Path Editor feature, you can list the untested paths of a specified function. Simply right-click on a function in a chart and select “Test Path Editor…” In the Test Path Editor window, check the option to “Show Only Untested Paths”.



If you click the “Show Test Paths …” button, the allowing you to scroll through each of the suggested paths and the corresponding sequence conditions needed to exercise the path. Adding tests that fulfill the conditions for each of the untested paths will achieve the coverage required for complete basis path testing.



Summary

This application note highlights how McCabe IQ’s cyclomatic path oriented technologies offer a unique and invaluable insight into security analysis. The example demonstrated how a control flow based approach is essential to assessing the exploitability of banned function usage and measuring the scope of attention required to remediate any security flaws.

McCabe IQ’s security analysis combines principles from structural analysis and quality analysis in an integrated tool, to help you make your software more secure.

For more information, or to schedule a demonstration of McCabe IQ, contact us at 800-638-6316 (in the US) or 401-572-3100. Visit us online at www.mccabe.com.