Analyzing Attack Surface Code Coverage

GCIH Gold Certification

Author: Justin Seitz, jms@bughunter.ca

Advisor: Joey Niem, detoor@gmail.com

Accepted:

Outline

1. Introduction

The art of analyzing a software system for security and robustness flaws can be a daunting task, and often begs a question: when is the analysis complete? Commonly a researcher or analyst answers this question by determining whether they have run out of budget, time, or have found bugs. However, these are not empirical pieces of evidence, what is really required is to understand how much of the software that is attackable was exercised.

Through this paper I will illustrate a theoretical means to determining how much of the useable attack surface is exercised by an analyst. Using a method like this is far from failsafe, as covering all pieces of code does not mean that there are no bugs, however it acts as a way to measure how effective your testing process is. I will outline what the attack surface is, what aspects of the surface are relevant (what I call the “useable attack surface”), current code coverage techniques and a way to tie code coverage to an attack surface.

As most software systems that are in widespread use today are either networked client/server applications or file parsing applications the paper will be based on testing applications that fall into the networked class. The secondary reason is that the highest level of risk is exposed by networked applications which can be accessed remotely and require no user interaction for compromise. I will also focus on Windows applications for analysis;

however the technique can be applied to any operating system and architecture.

2. Attack Surface

A software system's attack surface is the subset of resources that an attacker can use to attack the system [1].It is important to differentiate the classic definition of attack surface and what I call the useable attack surface.

The classic attack surface definition assumes all entry points into the software system; this can include certain Windows registry keys, open handles to windowed objects, or command-line parameters. The classic definition is an excellent means to track robustness across the entire system and to determine to what level you are willing to grant access to certain code areas to un-trusted users [2]. However, from a realistic perspective there would be no business model that would accept additional development time to ensure that Windows registry keys should be filtered before being used, for example, as this would become a very time consuming and burdensome part of the development cycle. This is not to say that it should not be done, rather there are very few circumstances where it is absolutely necessary.

The model of the classic attack surface makes it difficult to extract useful code coverage information specifically for a surface. The notion of a "useable attack surface"

means that it is the portion of the attack surface that an attacker could use to crash the software, access sensitive information or gain access to the host machine remotely. It is these particular areas of code that we are interested in exercising as they pose the highest degree of risk. It is also independent of configuration, and specification [3], as we are testing the connected code to a particular input source.

In the case of a networked application we will focus our attention on the system's socket receive operations, and the resulting packet parsing routines. It is here that most vulnerabilities, with the highest degree of risk, are found. In order to measure the total code coverage of this particular area, we need to first determine what functions are responsible for receiving packets on the network, and how the resulting data is passed along to the internal routines of the software. We also need to fully understand what code coverage is and how to apply it in this circumstance.

3. Code Coverage

Code coverage is a metric used in software testing that describes how much of the code in a program has been executed or tested [4]. In most cases, code coverage is considered a white box metric, as it involves instrumentation of the source code. However, code coverage metrics can be applied in a black box testing scenario, which is the method

that will be discussed throughout the course of this paper. However, it is important for the reader to fully understand code coverage techniques.

4. Testing and Code Coverage Methods

3.1 White-Box Testing

White-box testing involves the tester having internal knowledge, and source code of the application. This gives the tester the ability to "see inside the box", allowing them to fully understand the internal workings of the system [5]. In a typical QA environment this means that you have access to the source tree, and you are able to not only statically analyze the source code but also build test cases around the known paths inside the software. White-box testing is generally accepted as the primary method a QA team tests a product internally.

Using a white-box testing methodology allows for the testing team to write test cases based on units or components of the system that need to perform a certain task or function. It also allows for a very high resolution overview of code coverage, stability, resource usage and overall quality.

3.2 White-Box Code Coverage

White-box code coverage is the method of collecting a line-by-line code execution metric. Most code coverage tools are integrated into the development environment, and

provide a detailed analysis of what areas of code have been hit and what lie untouched. The white-box code coverage metric is useful for both QA and development teams to determine refactoring points, areas of code that can be removed as well as areas of code that can be improved in terms of performance. Although it is a useful metric for internal teams, it does not indicate the likelihood that a bug will be present in any particular area but it does provide excellent insight into the software the team is producing.

In white-box code coverage there are five primary metrics that can produce an overall code coverage number: function coverage, statement coverage, condition coverage, path coverage and entry/exit coverage. In function coverage the tester is try to determine how many functions have been exercised. Statement coverage involves a line-by-line analysis of how many lines of code have been executed. Condition coverage is a measure of how many decision points have been exercised. For example in a code block if there is a statement to test whether a number is higher than 0, condition coverage would be achieved with test cases of -1,1. Path coverage is determining if every route through the application has been covered and entry/exit coverage measures if every call and return from a function has been executed.

It is important to determine what type of coverage that you are attempting to achieve. For example, the following code example from Wikipedia best illustrates this:

```
void foo(int bar)
```

```
{
    printf("This is ");
    if (bar <= 0)
    {
        printf("not ");
    }
    printf("a positive integer.\n");
    return;
}
```

If function “foo” was called with “1” as its parameter we would achieve 33% condition coverage but only 80% statement coverage, as we would not hit the “printf (“not “);”. With a parameter of -1 we would then achieve 100% statement coverage but only 33% condition coverage. To achieve full statement and condition coverage we would then have to use three test cases: foo(1),foo(0),foo(-1). This is a very simplified example but it shows that white-box testing must have a clear target for a specific class of coverage in order to be effective.

It is helpful for the reader to understand that throughout the rest of this paper, we will be exploring *function level* coverage from a black-box perspective. There are a multitude of reasons why we are determining coverage in this way, all of which will be explained in the next section.

3.2 Black-box Testing

Black-box testing is the method whereby the tester has no knowledge of the inner

workings of the software they are testing [5]. However, the tester generally has a known specification for testing the software, and can determine correct and incorrect behavior based on that specification. Black-box testing is a necessary method to apply inside a QA environment as it is generally much faster than testing at a white-box or source level, while still having access to necessary internal information such as log files, error reports, etc.

In the case of vulnerability researchers, they are forced to approach the software system in the purest of black-box forms, as they have no inner knowledge, and no access to source should they need it. For the purpose of this paper it is essential to take this perspective in order for black-box code coverage to be utilized correctly.

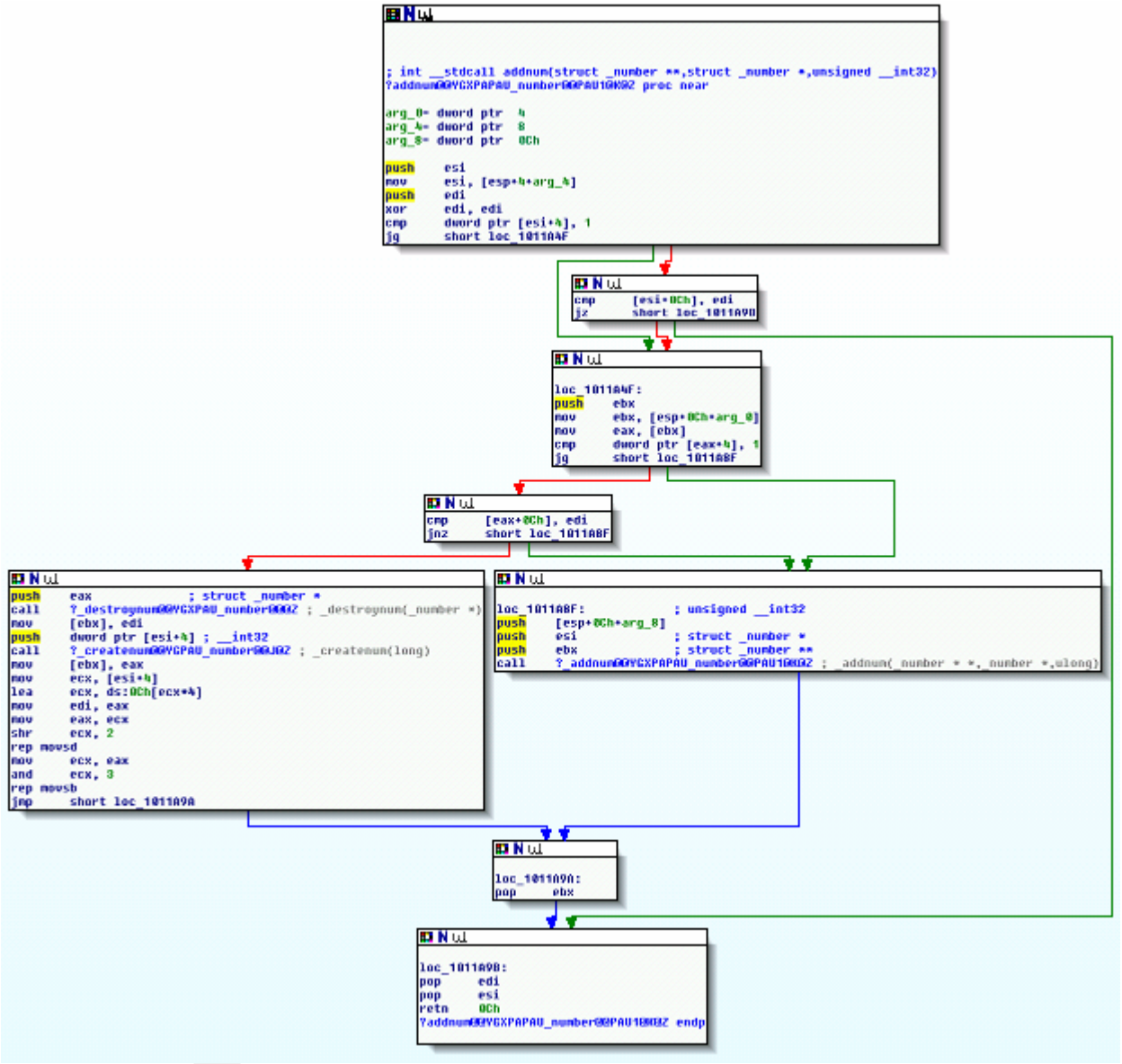
3.3 Black-box Code Coverage

Typically black-box code coverage is handled at the assembly level, and is completely outside of the normal internal testing cycle. No tools currently exist, or are necessary, inside of a development IDE that facilitate black-box code coverage, as one can get a much higher resolution of coverage from a white-box perspective.

In order to understand black-box code coverage, one must first understand how a binary is broken down from an assembly perspective. Inside of each binary's code segment, there are functions and basic blocks. Functions can be thought of just as their C

counterpart: you give parameters to a function, you call the function and it gives a return value. There is not any difference from an assembly perspective either, although there are subtle differences in how functions are called in some compiler implementations, but this is beyond the scope of this paper.

Each function is comprised of one or more basic blocks. Basic blocks are small portions of the function which are terminated by a branch instruction, a call or a return [6]. In Figure 1. you see the addnum() function from the default calculator shipped with Windows XP.



(Fig. 1. - IDA Pro 5.1 Representation of a function and its basic blocks)

The top node in the graph is the head of the function, and you can see it takes in parameters just like in C. Each of the nodes below it are the basic blocks of the function.

Black-box code coverage is measured in two different ways: function level coverage and basic block level coverage. Using a function level coverage routine, the tester would determine the addresses of each of the function heads inside of the binary, and then set breakpoints on each address. Every time a breakpoint is hit, it is deemed a code coverage hit, and then execution is allowed to continue [7]. Using this method, you are able to determine a fairly high-level measure of the code coverage during a testing run. The method for measuring basic block level code coverage is no different, however one must first find all of the functions, and then determine all of the function's basic blocks, and set the breakpoints on the basic block heads. This provides a much higher resolution code coverage metric, but also is a much more lengthy process as each breakpoint interrupts the processor on the tester's machine, and there can be tens of thousands of basic blocks in even the simplest applications.

Using our previous example for code coverage, where we had the function "foo()" which accepted an integer as a parameter, it produces the following disassembly using IDA Pro 5.1:

```
0x00401000 sub_401000
0x00401000 arg_0 = dword ptr  8
0x00401000 push    ebp
0x00401001 mov     ebp, esp
0x00401003 push    offset aThisIs  ; "This is "
0x00401008 call    _printf
0x0040100D add     esp, 4
0x00401010 cmp     [ebp+arg_0], 0
0x00401014 jg      short loc_401023

0x00401016 push    offset aNot     ; "not "
0x0040101B call    _printf
0x00401020 add     esp, 4

0x00401023 push    offset aAPositiveInteg ; "a positive integer.\n"
0x00401028 call    _printf
0x0040102D add     esp, 4
0x00401030 pop     ebp
0x00401031 retn
```

For this simple case, we can cross-reference the disassembly to the source code easily. We see the function begins at 0x00401000, and that IDA has detected it takes a single parameter (“arg_0 = dword ptr 8”). We can also clearly see the first printf() being called exactly as it is shown in the source code.

The interesting part of the code begins at 0x00401010 where we see a “cmp [ebp+arg_0],0”. In pseudocode this means “compare the value of the function parameter to zero.” The CMP instruction in x86 assembly will set the zero-flag register depending upon the evaluation. If the comparison is true (in our test case 0 or -1), it will set the ZFlag register to 1. Likewise, if the comparison is false (in our test case 1), it will set the register to 0. The next

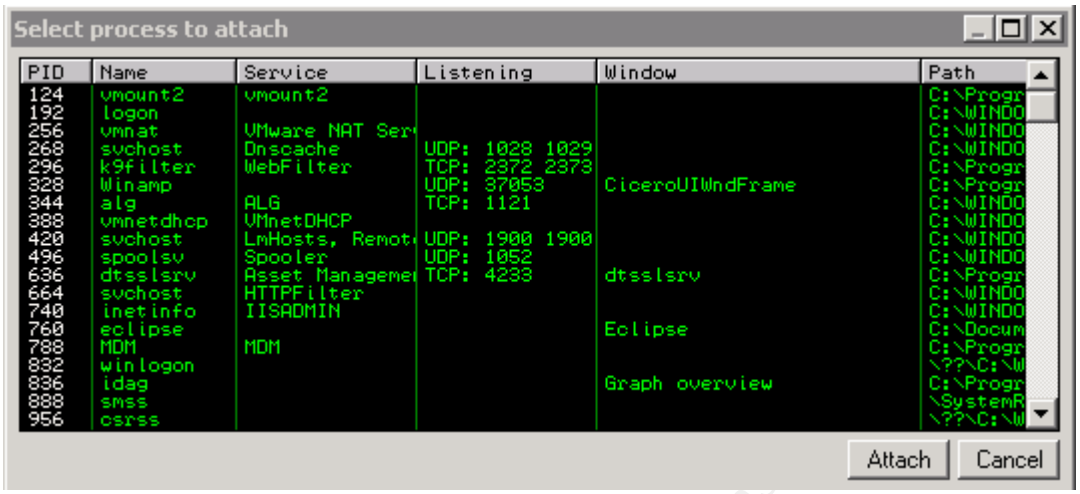
instruction at 0x00401016 “jg short loc_401023” tests whether the CMP instruction evaluated to true or false by checking the ZFlag register. In pseudocode this would be “if the comparison was true then continue executing, if the comparison was false then jump to the next printf() call.”

We can now see and understand the translation between what we see as source code, and what we see in a black-box or disassembled format. Again, this is a very simple example, but as the code complexity increases the gap between white-box and black-box widens and it becomes more difficult to determine how to achieve 100% decision, statement, path and entry/exit coverage. For example, in our source code, we really only have 5 lines of executable code inside of the function foo() but the disassembly contains 14 lines of executable code, this makes it difficult to accurately use statement coverage in a black-box fashion. This is also why we choose function-level coverage, as we can easily decode the beginning of a function, and measure when we have hit it by using breakpoints. In this case we would set a breakpoint on 0x00401000 and when that breakpoint was hit we know that function foo() had been executed.

As a side note, one could also pull in the functions and basic blocks for an application, and using some simple heuristics determine decision coverage by the number of basic blocks that were executed based on whether the basic block was reached by a conditional jump, but this is a very slow and painstaking process, and does not lend itself to quickly measuring the amount of code attached to an attack surface.

5. Attack Surface Determination

For the purpose of this paper we will determine the attack surface of an application, based on the network port(s) that it is listening on. Using a tool called Immunity Debugger, we can easily see what port a process listens on (see Figure 2).



(Fig. 2 – Immunity Debugger's view of all running processes)

This makes it very easy for us to attach to a process and know that we have a particular port open and awaiting input. In order to make use of that particular piece of information, we need to understand what an open and listening port really means at the binary level, and where we want to monitor code coverage.

4.1 Hooking Win32 Socket Operations

In order for us to focus solely on the network level attack surface, we have to begin trapping code coverage metrics at the point when a packet has been received and is traversing memory into the application's logic. There are four standard Windows socket

routines that are generally used in networked applications: recv() (TCP), recvfrom() (UDP), WSARecv() (TCP) and WSARecvFrom() (UDP). All four of these functions are exported from the system library WS2_32.dll which is located in the C:\WINDOWS\system32\ directory.

To hook any of these functions, we are merely setting a breakpoint at the function head of those calls and then monitoring where the function returns. The return point of the function will be the point at which we want to begin monitoring code-coverage, as it will be the main application that has made the call to the receive functions in order to receive and process packets from the network.

The addresses of both the call to the receive operation and the return from it are easily accessible in IDA and Immunity Debugger, just by searching for the socket operation function name and setting a breakpoint. If you are doing the hooking at runtime, then it is important to set two hooks: the first hook is at the head of the receive operation, when that hook gets hit, you then set a hook on the stack pointer ([ESP]) which points to the return address of the calling function. You are now prepared to begin tracking code coverage from the point the packet has been received off the network, and is about to be processed by the application.

6. Measuring Attack Surface Code Coverage

5.1 Determining Overall Code Area

To begin a code coverage run, it is generally useful to acquire the total number of functions and basic blocks contained within the application. After we have determined the overall code area, we can then move on to determining what portion of the code are is the attack surface we are looking to analyze. Figure 3 depicts the usage of the Immunity Debugger's ability to analyze a binary and retrieve a total function count:

```
*** Immunity Debugger Python Shell v0.1 ***
Immlib instanciated as 'imm' PyObject
READY.
>>>main_module = imm.getModule(imm.getDebuggedName())
>>>imm.analyseCode(main_module.getCodebase())
>>>
>>>func_list = imm.getAllFunctions(main_module.getCodebase())
>>>print len(func_list)
145
```

(Fig. 3 – Immunity Debugger pyShell depicting a function count)

From this small script we are able to see that the currently running process has 145 total functions that can be reached. To achieve code coverage on the binary as a whole, we would then iterate that list and set a breakpoint on each address. Keep in mind this is only the coverage information for the primary executable, if it includes other dynamic libraries (in the

case of Win32 these are DLLs) that are not system flagged, then we would have to make sure to analyze and set breakpoints inside those libraries as well.

5.2 Determining Attack Surface Code Area

In order to determine the code area of the attack surface, we have to recursively determine all code cross-references stemming from the reception of a packet. An illustrative example would be this:

1) Address “A” is the return address from a WS2_32.recv call.
2) Determine the cross-references to the function where address “A” resides. Let’s say that there are three function addresses that are cross-references, we will call them X, Y, and Z.
3) Now begin recursively determining the cross-references to X,Y and Z, thus building a list that would look like [X1,X2,X3…], [Y1,Y2,Y3…] and [Z1,Z2,Z3…]. For each of the items in the resulting lists we then have to determine the cross-references to them, and so on.
4) Adding up all of the cross-references gives us the total function count that is related to the attack surface we are analyzing.

Now that we have determined how many functions comprise the attack surface we are able to again simply add breakpoints for each of those addresses. We have completely reduced the amount of code coverage area that we have to monitor. This is a very targeted set of coverage points, as opposed to having to manually filter out extraneous function calls which may not be related to the handling of received network data.

5.3 Fuzzing for Attack Surface Coverage

Generally, code coverage data is useful from the perspective of a tester when they want to ensure that they don't have any extraneous code that needs re-factoring or is no longer used. From the perspective of a security engineer, code coverage is more interesting for measuring the effectiveness of a fuzzing run. One can measure the effectiveness of their fuzzing tool not only by the number of bugs it finds, but also on the amount of code coverage that was exercised during a fuzzing run. Again, the main caveat is that 100% code coverage does not ensure there are no bugs.

To gain an understanding of the code coverage before and after 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 coverage we are able to determine the common path that a

normal packet will take through the application's logic. Once we have determined the code coverage count from a regular packet, we can then begin mutating that packet to see if we can begin exercising other chunks of code that are not normally executed, for example error handling routines, or logging functions. From personal experience, most developers put a lot of time and effort into making sure the packet parsing is done correctly, but when displaying an error message will blindly pass the user-supplied data into a buffer for display, which is generally a bad idea.

5.3.1 Genetic Fuzzing and Code Coverage

The newest form of fuzzing that has begun to take hold is genetic fuzzing. Genetic fuzzing is the method of testing whereby the inputs are generated based on a genetic algorithm that uses various metrics (for now only code coverage) to determine the fitness level of the input being sent to the application. This enables the fuzzer to become "smarter" over time and to provide inputs that cover greater amounts of code over each new generation.

An excellent example of a genetic fuzzer is EFS [8]. EFS fully utilizes a reverse engineering framework called PaiMei to trap code coverage data on each packet that is sent to the target application. It then uses this code coverage information to create the best packets that will drive the deepest into the application's logic, and this has been proven to be a very powerful method for robustness testing a software application.

An interesting way to make the EFS system even more powerful would be to apply our attack surface code coverage method to the PaiMei framework, which is responsible for the function resolution and breakpoints. This would generate a very small subset of the total code coverage paths and EFS would then have a fine-grained list of coverage points that would only be relevant to the attack surface we wish to test. This would also alleviate the current problem where you have to do pre-test runs using PaiMei to manually determine what code coverage hits should be filtered out before EFS fires up.

7. Attack Surface and Incident Handling

Typically, an incident handler is not working in close association with the QA or development teams in terms of testing metrics, coverage and other software development lifecycle areas. In the case of attack surface, it can be useful for an incident handler during their preparation phase to be able to determine what an application's attack surface looks like (what network ports does it listen on, does it read environment variables, etc.). It can also be very useful for a handler to have an idea of how well the development team has exercised the attack surface, and to understand where they can obtain further information about the application's health should an attack or incident occur.

We may soon see the gap being bridged between incident handlers, application testers and development staff. It is crucial that there is no knowledge gap between all teams, and the attack surface of deployed applications is a very relevant nugget of information that should be shared.

8. Conclusion

We have demonstrated an easy, and repeatable method for not only determining the amount of code attached to a particular attack surface, but also how to practically use that information as a testing metric to find bugs in software applications. The attack surface itself is a difficult to define part of an application, but using the approach as outlined in this paper, we can begin honing in on only the most interesting (and risky) parts of an application.

In the future, we should expect to see fuzzers more heavily relying on code coverage metrics, and the black-box methods may soon overtake the white-box methods. Software security testing is an interesting and exciting field, and I hope that there is a continued effort to develop smarter methodologies for finding bugs.

9. References

[1] Manadhata, P. K., Wing, J. M., Flynn, M. A., & McQueen, M. A. (2006). Measuring the Attack Surfaces of Two FTP Daemons. 2.

[2] Howard, M. (2004). Mitigate Security Risks by Minimizing the Code You Expose To Untrusted Users. Retrieved September 30, 2007, from Attack Surface: Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users Web site: http://msdn.microsoft.com/msdnmag/issues/04/11/AttackSurface/

[3] Guruswamy, K. (2007, Jan 14). Attack Surface Coverage. Retrieved October 01, 2007, from Attack Surface Coverage Web site: http://labs.musecurity.com/2007/01/14/attack-surface-coverage/

[4] Code coverage. (2007, October 26). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:44, October 30, 2007, from http://en.wikipedia.org/w/index.php?title=Code_coverage&oldid=167310512

[5] Patton, R. (2000). *Software Testing*. Sams Publishing.

[6] Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. Wiley Publishing.

[7] Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison Wesley.

[8] DeMott, J. (2007, Aug 1). Evolutionary Fuzzing System. Retrieved September 30, 2007, from Evolutionary Fuzzing System Web site: http://vdalabs.com/tools/efs.html