

Static Analysis of Anomalies and Security Vulnerabilities in Executable Files

Jay-Evan J. Tevis
Department of Computer Science
Western Illinois University
Macomb, Illinois 61455
(309) 298-1035
jj-tevis@wiu.edu

John A. Hamilton, Jr.
Department of Computer Science
Auburn University
Auburn, Alabama 36849
(334) 844-4330
hamilton@eng.auburn.edu

ABSTRACT

Software researchers have already developed static code security checkers to parse through and scan source code files, looking for security vulnerabilities [8, 9]. What about executable files? Can these files also be statically checked for security weaknesses such as buffer overflows? We have created a methodology that uses information located in the headers, sections, and tables of a Windows NT/XP executable file, along with information derived from the overall contents of the file, as a means to detect specific anomalies and software security vulnerabilities without having to disassemble the code. In addition, we have instantiated this methodology in a software utility program called findssv that automatically performs this static analysis.

We tested findssv on six categories of files: executable installation files, software development files, Windows XP operating system files, Microsoft application files, security-centric applications files, and miscellaneous application files. Through the test results on over 2700 files, we show that findssv can detect 1) inconsistent table sizes, 2) large zero-filled regions of bytes, 3) unknown regions of bytes, 4) compressed files placed in a file, 5) sections that are both writable and executable, and 6) the use of functions susceptible to buffer overflow attacks. We also identify key vulnerability findings about the software in the six categories.

Categories and Subject Descriptors

D.1.1 [Software Engineering]: Testing and Debugging.

D.4.6 [Operating Systems]: Security and Protection – invasive software.

General Terms

Languages, Security.

Keywords

Software security vulnerabilities, static analysis, executable file, PE format.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SE'06, March 10-12, 2006, Melbourne, Florida, USA
Copyright 2006 ACM 1-59593-315-8/06/04...\$5.00.

1. INTRODUCTION

New computer viruses threatening our business and home computers have become a regularly expected occurrence. Moreover, we have come to accept the installation of a software patch as the preferred means to stop such malicious software. Companies have devoted significant resources to anti-virus teams and defense strategies to combat these problems. Such strategies include firewalls, intrusion detection mechanisms, honey pots, port monitors, system security scanners, and internet/e-mail content scanners [3]. Instead, we should look at ways to build more secure software from the start and identify methods to detect vulnerable software applications before installing them on our computers.

Security researchers have developed many software tools that provide a security check of C and C++ source code [8]. The tools search for functions and data structures that pose a risk to the security of a computer system. These insecure items can be considered a doorway through which viruses and other malicious code enter to attack a system [6]. The security checkers search for these "doorways" (i.e., vulnerabilities) in a source code file and alert the programmer to their presence.

Though these security checkers provide some assistance in the prevention of security vulnerabilities, they only concentrate on source code. The hackers, on the other hand, target the vulnerabilities in the executable code. In general, these are the same vulnerabilities that secure programming practices should strive to prevent. By disassembling an executable program and looking for certain key indicators, hackers apply their tools of exploitation.

Our research effort involves the scanning of executable files for software security vulnerabilities. Specifically, it involves the scanning of files that conform to the portable executable (PE) format designed for software running on Windows NT/XP computers. We set out to show that a methodology can be devised that uses information located in the headers, sections, and tables of an executable file, along with information derived from the overall contents of the file, as a means to detect specific anomalies and software security vulnerabilities without having to disassemble the code. We also set out to instantiate such a methodology in a software utility program that automatically detects certain software security vulnerabilities without installing and running the executable file.

This paper contains a summary of our approach and findings. ([7] contains the details of this research effort.) We began by identifying specific information in the PE format that is useful in a

security vulnerability analysis. We then formulated a methodology for identifying certain security vulnerabilities using this information. We incorporated this methodology into a software application called **findssv** that dissects a PE file and analyzes its parts. We ran findssv on six categories of executable files (over 2700 files in all). Based on the test results, we gathered together our key findings and drew conclusions on the usefulness of this static analysis approach.

2. THE PE FORMAT FROM A SECURITY POINT OF VIEW

The typical items in a PE file are the DOS header, the MS-DOS stub, the PE signature, the file header, the optional header, the section table, the symbol table, the string table, various sections (e.g., ".text" and ".data"), the import table, and the export table. The `winnt.h` file, which is a standard C include file for Windows programming, describes the byte format for each of these items by means of type definitions.

The following paragraphs briefly describe each typical item found in a PE file and point out any indicators of anomalies or security vulnerabilities that can be obtained from analyzing these items. For a more detailed explanation of the PE format, refer to [5].

2.1 The DOS Header

The DOS header is a single record of data located at the beginning of each file. It contains information used mainly by the loader program. For example, one field in the record contains the offset for the start of the PE signature field that appears later in the file. We found no information in the DOS header that was useful for detecting security vulnerabilities.

2.2 The MS-DOS Stub

The MS-DOS stub follows directly after the DOS header and is a holdover from the days when PC software only ran in the MS-DOS operating system. If a PE file is invoked from the MS-DOS prompt, and that program requires the Windows GUI mode to run, then the MS-DOS stub displays a default message stating that the program cannot be run in MS-DOS mode. The program then terminates.

We found no information in the MS-DOS stub that was useful for detecting security vulnerabilities. However, between the end of the MS-DOS stub and the location in the file where the PE signature field is located, there should be a contiguous region of zero-filled bytes. The loader ignores any information in this area; consequently, software developers might use this area to store application-specific messages or data such as a cryptographic key. Although we see no direct security risk in this information, we consider any non-zero data in this region as an anomaly in the file.

2.3 The File Header

The start of the file header follows the PE signature field. (The PE signature simply contains the letters "PE".) The file header is a single record that reveals much about the operating environment that the file expects, the contents of the file, and the characteristics of the file. It tells the number of sections in the file, the start location of the symbol table, the number of symbols in the symbol table, and the size of the variable-length optional header. It also implicitly tells the start location of the string table because that table directly follows the symbol table.

We found no information in the file header that was useful for detecting security vulnerabilities. However, the file header does reveal the presence or absence of the symbol table and string table. As we will see later, these two tables are very helpful in detecting buffer overflow security vulnerabilities in a file.

2.4 The Optional Header

The start of the optional header immediately follows the end of the file header. The name of the optional header is misleading. It is only optional in the common object file format (COFF) used in an object code file; it is mandatory in the PE file format. Except for the last field of the optional header, the rest of the header is a fixed record size. The data fields in this record reveal even more detail than the file header does about the operating environment expected and the runtime requirements required by the file. However, we found no information that was useful for detecting security vulnerabilities.

The last field of the optional header is referred to as the data directory. It is a variable-length list of records. Each record contains the start location and size of a data table located in the file. For example, the location and size of the import table and export table are found in this list.

We found no information in the optional header that is useful for detecting security vulnerabilities. However, we do compare the stated size of each table in the data directory (such as the import table) to the actual size that we detect later when reading the contents of the table in the PE file. If there is a difference in these two sizes, we consider this a file anomaly. We also consider it an anomaly when the data directory lists an optional table that does not actually appear in the file.

2.5 The Section Table

The start of the section table immediately follows the end of the optional header. It contains one or more fixed-sized records with information on the start location, size, and characteristics of each section in the PE file. The number of entries in the section table is denoted by the value in the Number of Sections field in the file header. The section entries are listed in the order that the sections appear in the PE file.

A security vulnerability can be detected in the section table by looking at the characteristics for a section. These characteristics tell the loader program if the contents of the section contain executable code or initialized/uninitialized data. They also tell if the section's contents can be executed, read from, or written to. Each of these characteristics has its own bit position in a 64-bit word. To be secure, any section that has the characteristic "contains executable code" or "can be executed" bits set should not have the characteristic "can be written to" bit also set. If such a situation occurs, the program will have the ability to modify its own executable code in memory at run time.

A security vulnerability can also be detected in a PE file by looking for 1) a section entry that is located in the section table but does not have a corresponding section in the file or 2) a section entry in the section table that exceeds the number of sections denoted in the file header. Although both of these occurrences could be considered anomalies, they could also indicate that a virus has tampered with the contents of a file by deleting or adding a section.

2.6 The Symbol Table

The size and location of the symbol table are based on two fields in the file header that indicate this information. If the value in the Number of Symbols field is zero, then the file contains no symbol table. The linker uses the symbol table in an object code file, but the table is not required in a PE file. The table is only present there for the use by a debugger. The symbol table contains the name of every data item and function declared by the program and by any library or runtime environment code linked in with the program. Depending on which linker is used, the symbol table also contains the names of source code files for the program.

A search of the function names in the symbol table can be used to detect a buffer overflow security vulnerability in a program. Each function name listed in the symbol table can be compared to a list of standard C library functions susceptible to buffer overflows [9]. The resulting matching entries reveal vulnerable functions that were either used in the application source code or were injected into the object code by the compiler during source code translation.

2.7 The String Table

The string table is located immediately following the symbol table. If a symbol table is not present in the file, then the string table is located where the symbol table would have started. If the start location of the symbol table field in the file header is zero, then the file does not contain a string table. The linker needs a string table in an object code file, but the table is not required in a PE file. As with the symbol table, the string table is only present there for use by a debugger. The string table contains the name of every variable name and section name that appears in a program or in any library or runtime environment code linked in with the program. The section table and symbol table only contain entry names up to eight bytes in length. For a name of any greater length, an offset is stored in the name field instead. This offset points to the start of a character string in the string table.

Although we found no information in the string table that was directly useful for detecting security vulnerabilities, the contents are helpful indirectly in supplying the names used in the section table and symbol table.

2.8 Various Sections

The sections referred to in the section table complete the remainder of a PE file's contents. Some linkers place each optional table in its own section. For example, the export table may be in the ".edata" section. Other linkers place the optional table in the midst of other data in a section. For example, the import table may be in the ".text" section. The actual executable code and constant data we commonly associate with an executable file are stored in one or more sections. These sections are usually referred to, respectively, as the ".text" and ".data" sections, although these may have other names. Because we do not disassemble any executable code, we found no information in the sections themselves that was useful for detecting security vulnerabilities.

2.9 The Import Table

The location of the import table in a PE file is found by calculating the file offset of the starting location. This is done using the Relative Virtual Address field in the data directory of the optional header and the Virtual Address and Pointer to Raw

Data fields in the section table. The import table contains the names of the dynamic link library (DLL) files and the functions in those files that the program requires in order to run. If a program does not use any DLL files, then the import table is not present.

As we described earlier concerning the symbol table, the import table also contains the names of functions used in the file, albeit only functions imported from DLLs. This information can be used to determine a buffer overflow security vulnerability in a file similar to the method used with the symbol table.

Note that if a file uses no DLLs and also contains no symbol table, then a search for the names of vulnerable functions will come up with an empty list. This should not be considered an indication that the program uses no vulnerable functions. Instead, it should point out the lack of enough information to make such a determination.

Although the use of a specific DLL by a program does not, in itself, constitute a security vulnerability, a security analyst may be interested in knowing about the DLLs used by a file. Looking at the purpose or function served by a DLL may reveal operations performed by the program that aren't readily apparent from the user interface or software documentation. This could include DLLs to assist in network connections, encryption, or installation of files.

2.10 The Export Table

The location of the export table in a PE file is found by calculating the file offset of the starting location as was described above for the import table. The export table contains the name of each function in the file that is available for use by another program. This is the normal behavior for a DLL; consequently, an export table is usually not found in other types of PE files. We found no information in the export table that was useful for detecting security vulnerabilities.

3. A METHODOLOGY FOR DETECTING ANOMALIES AND SECURITY VULNERABILITIES

This section describes our methodology for statically analyzing a PE file to provide information on anomalies and security vulnerabilities. When analyzing the file, the methodology categorizes its findings as facts, anomalies, and vulnerabilities. The methodology in the following paragraphs is organized around these three categories.

3.1 Creating a File Fact Summary

Based on the administrative information stored in the DOS header, file header, and optional header, we generate a file fact summary. These facts include the actual file size in bytes, the target CPU and operating system, an indication that the file is a DLL file or not, an indication that the file has a symbol table or string table, the names of the text files containing the program source code, and a list of optional tables found in the file. By examining the import table, we build a fact list of the DLLs required by a program.

3.2 Detecting Anomalies When Reading the File

As alluded to in subsections 2.2 and 2.4, we detect anomalies in the process of reading a PE file. One interesting anomaly is the

mismatch between the size of a table as stated in the optional header and the actual size of the table in the file. Such anomalies may indicate an error in the linker or possibly manual tampering with the file contents.

3.3 Detecting Anomalies When Mapping the File Contents

Another interesting anomaly is the detection of regions in the file that contain either unknown information or compressed files used in a software installation. We detect this by mapping the contents of the bytes of a file from the first byte to the last. As each header, table, or section is read from the file, we record its start and stop location (i.e., map it). After mapping the locations of every known region in a PE file, we make a complete pass through the map to find byte regions that are unaccounted for. We mark such a region as "contents not known" if it falls outside the bounds of any known region in the file. Otherwise, we mark it as "no additional details" if it is inside a section containing an optional table but not also inside the bounds of the table itself.

We detect compressed files by comparing an unknown region's size to a maximum acceptable size constant. Based on a survey we made of the average maximum size of known regions in a variety of PE files, we found that 200,000 bytes is a good constant. During that survey we also found that large unknown areas occur in executable installation files where the files to be installed are stored in a compressed format in the executable file. These areas occur most often just before or directly after the last known section in a file. Some also occur as part of a ".rsrc" section or a ".winzip" section. The loader ignores any bytes in a file that are not part of one of the listed headers, sections, or tables; consequently, an executable installation file can act as its own repository of hidden data.

3.4 Detecting Software Security Vulnerabilities

Our research has revealed three kinds of security vulnerabilities that can be detected when statically analyzing a PE file: 1) sections in a file whose contents can be both written to and also executed, 2) large unused zero-filled regions in a file, and 3) the use of functions susceptible to buffer overflow attacks. In the following paragraphs, we describe the means for detecting these vulnerabilities.

3.4.1 Detecting Sections that are Both Writable and Executable

Earlier in this paper when we described the section table, we pointed out that each entry in the table contains a 64-bit characteristics field. Bits in this field indicate the read, write, and execute characteristics of the contents of the corresponding section. We detect a security vulnerability by checking for the simultaneous occurrence of both the write and execute characteristics for a section. Such an occurrence may indicate an error in a linker. It may also indicate tampering by malicious or obfuscation software in an effort to later modify the program's executable code when it is loaded into memory and executed.

3.4.2 Detecting Large Unused File Regions

When mapping the regions of a file, we track the zero-filled regions and maintain a total of the number of zero bytes that are encountered. Based on a survey we made of the average size of

zero-filled byte regions in a variety of PE files, we found that a value less than or equal to 50 is acceptable. Any value above this constant indicates a vulnerability in which malicious software can use these areas to store hidden code or data.

3.4.3 Detecting Vulnerable C Library Functions

Earlier we described how examination of the symbol table and the import table could reveal the names of many of the functions used by a program. [9] provides a list of commonly-used C library functions that are vulnerable to buffer overflow attacks on their character string parameters. When analyzing a file, we build a list of vulnerable functions used by a software program. We do this by comparing the function names found in the symbol table and import table to those found in the list of known vulnerable functions.

3.4.4 Understanding the Consequences of no Symbol Table or Import Table

The ability to detect the use of functions vulnerable to buffer overflow attacks obviously depends on the presence of the symbol table and/or the import table in the PE file. If the software developer passed an option to the linker to strip the symbol table when the executable file was built, then that source of information does not exist in a PE file. Also, if the linker placed the function definitions in the executable file rather than arrange to have them be accessed through a DLL, then the import table will not contain the names of the vulnerable functions. (In addition, an import table may not even be present in the file.) This finding is very important to understand. Without it, a person can get a false sense of security if no vulnerable function calls are found when statically analyzing a PE file using this approach. In other words, an empty list of vulnerable function names can only safely indicate the lack of enough information to detect any function names at all.

4. AUTOMATION OF THE METHODOLOGY: THE FINDSSV SOFTWARE UTILITY

From a practical point of view it would be very difficult to manually test and utilize the methodology we described in Section 3 for even the smallest of PE files. This is because of the tens of thousands of bytes in PE administrative information and runtime environment code included in the file. Instead, we automated the methodology by developing a software utility program called "findssv", where "ssv" stands for software security vulnerabilities. Its purpose is to assist a software security analyst in performing a static analysis of any executable files and dynamic link library files.

4.1 Program Description

Findssv is written in C++ and consists of 3800 source lines of code. It operates similar to an MS-DOS command line utility and is designed to detect anomalies and certain software security vulnerabilities in PE files that run on Windows NT. It also works on object code files that use the Microsoft Common Object File Format (COFF). Findssv accepts an executable or object code file name (or a wildcard form of a file name) on the command line, followed by zero or more options. It displays information about each of the PE and COFF files in the form of a report that is sent to the standard output device (i.e., the screen). It can be used to

automatically check for anomalies and security vulnerabilities in a single file or in a whole directory of files in just a few seconds. In addition, it can be used to create a textual map of the file layout.

4.2 The File Mapping Feature

The file mapping feature of findssv collects, analyzes, and displays the byte-for-byte layout information of a COFF or PE file. The map output shows the byte location and size of every header, section, and table in the file. It also points out the areas of the file whose contents are unknown or contain only a series of zeros.

Findssv stores the file map information in a map data structure. The key for each entry in the map is the start address of a region (i.e., a header, section, or table) in the file. The corresponding data part of each entry contains the region's size and a short description of its contents.

Findssv uses a set of algorithms to locate and map unknown data regions in a file, to locate and map the zero-filled regions in a file, and to display the layout of all of the regions in a file in a hierarchical format. The hierarchical format is necessary because some regions of a file may be contained within other regions. For example, the ".text" or ".data" section of a file may contain the import table along with other information.

4.2.1 Locating and Mapping Unknown Regions

The algorithm to locate and map unknown regions looks for byte ranges in the file that are unaccounted for in the file map. Findssv performs this action after all of the known regions of a file have been read and mapped. This process involves the tracking of the start and stop addresses of large regions in a file that may contain smaller regions. After findssv detects an unknown region, it enters the region in the file map. If the region is outside of any other region, it describes the region as "Contents not known"; otherwise, it describes the region as "No additional details". This is because it occurs inside an already-mapped region.

4.2.2 Locating and Mapping Zero-filled Regions

The algorithm to locate and map zero-filled regions looks for byte ranges in the "Contents not known" regions of a file that contain a program-defined minimum number of consecutive zero bytes. Consequently, this algorithm is run after the algorithm that locates and maps unknown regions. When findssv detects a "Contents not known" region exceeding the minimum size, it inserts this information into the file map.

4.2.3 Displaying the File Map Layout

The algorithm to display the map layout of the file takes its information directly from the contents of the file map structure. It keeps track of any smaller regions that are mapped inside of any larger regions. The resulting map display gives a security analyst a revealing view of how the contents of the file are ordered and structured. It also reveals areas, such as the regions marked as "Contents not known", that may require further investigation.

5. RESULTS FROM TESTING THE AUTOMATED METHODOLOGY

Our test platform consisted of a Hewlett-Packard personal computer with a 1.3GHz Intel Celeron processor, 512MB RAM, 40GB hard drive, running the Windows XP Home Edition operating system. The test files consisted of 2725 PE files either

already pre-installed on the computer or used frequently for home office use, computer science research, personal entertainment, or software development.

5.1 Three Test Objectives

We had three test objectives. Our first objective was to determine that findssv could detect the anomalies and security vulnerabilities that we had identified in our methodology described in Section 3. Our second objective was to determine if findssv could correctly read a vast assortment of PE formatted files. Our third objective was to determine if the automated methodology would produce meaningful and useful results on the anomalies and security vulnerabilities searched for by findssv.

5.2 Test Approach and Results

To achieve our first objective, we tested findssv on a set of specific example files. This also allowed us to fine-tune the findssv software and the automated methodology. In addition, it allowed us to see what facts, anomalies, and security vulnerabilities we could discover about software for which we had a special interest, such as findssv itself. To achieve our second and third objectives, we identified six categories of PE files: executable installation files, software development files, Windows XP operating system files, Microsoft application files, security-centric application files, and miscellaneous application files. [7] contains the details of the results we obtained from testing all of the files.

5.3 Key Findings from the Test Results

We gathered the following key findings after analyzing the test results produced by findssv:

- It is possible for an executable file to reveal less information to hackers about the functions it uses by 1) having its symbol table stripped and 2) by having the linker include the language's standard function definitions in the executable file rather than reference the functions in a dynamic link library.
- A program compiled and linked using Cygwin Gnu tools [2] will have standard C functions in it that are susceptible to buffer overflow attacks even when these functions are not explicitly used by the software developer in the source code files.
- The Cygwin Gnu C++ compiler and linker injected seven vulnerable function calls into the executable program of the findssv program.
- It may be possible to analyze the general layout of the sections and tables in a file map of an executable file in order to detect a pattern that indicates the compiling and linking tools used to generate the file.
- The Cygwin1.dll file contains security vulnerabilities that allow executable code to be modified after the program is loaded into memory and executed.
- The Kernel32.dll file contains functions that are susceptible to buffer overflow attacks.
- An executable or dynamic link library file can take advantage of the flexibility of the PE format and serve as its own storehouse for millions of bytes of data.

- The Cygwin software development files and utility programs contain scores of security vulnerabilities. Therefore, we do not recommend them for secure programming activities.
- 42 of the 56 Sun Microsystems Java software development files that we tested (including the Java interpreter) contained one or more functions that are susceptible to buffer overflow attacks.
- 8 of the 32 Microsoft Visual Studio SDK files [10] that we tested (including the MSIL disassembler) contained one or more functions that are susceptible to buffer overflow attacks.
- 13 of the 21 Microsoft Visual Studio C/C++ 7.0 [10] files that we tested (including the compiler and linker) contained one or more functions that are susceptible to buffer overflow attacks.
- In the “c:\windows\system32” directory of the Windows XP Home Edition, approximately 25% of the executable files and dynamic link libraries use one or more standard C functions that are susceptible to buffer overflow attacks.
- In the Network Associates Common Framework software installation that accompanies the VirusScan software installation, approximately 75% of the executable files and dynamic link libraries use one or more standard C functions that are susceptible to buffer overflow attacks.
- In the Zone Alarm Pro 4.0 software, all three of the dynamic link library files contain standard C functions that are susceptible to buffer overflow attacks.

6. RELATED WORK

We found no published research related to our work described in this paper. However, back in the fall of 2004 we did uncover information on two commercial products, SmartRisk Analyzer [1] and BugScan [4], that scan executable files for security information. Since then Logic Library bought BugScan from the product’s builders, HBGary. The software is now called Logiscan. Also since then, Symantec bought @stake, the company that developed SmartRisk Analyzer. We found no current information at the Symantec’s web site on the status of SmartRisk Analyzer. However, according to the documentation and reviews found on the WWW from 2004, both of these software tools perform in-depth security analysis of executable files. Except for advertising brochures and FAQ lists, we found no published information on the detailed methodologies implemented in these products. In addition, we found no test results demonstrating how their scanning approaches work.

7. CONCLUSION

Our main goal throughout this paper has been to describe a methodology for automatically detecting specific anomalies and security vulnerabilities in executable program files through static code analysis. Published open source auditing techniques only describe automated static code analysis of source code files. Because of our research effort and results, a second approach directed at executable files now exists in open source.

We showed that a methodology could be devised that uses information located in the headers, sections, and tables of an executable file, along with the information derived from the

overall contents of the file, to detect specific software security vulnerabilities without having to disassemble the code. In addition, we showed that the methodology could be instantiated in a utility program that detects certain anomalies and security vulnerabilities without installing and running the executable file.

8. FUTURE WORK

We believe there is yet more security-related information to be found in executable files through static analysis. Further research is needed to identify more key indicators of software security vulnerabilities that can be detected by automated means. Consequently, we plan to pursue the following initiatives:

- Determine the compiler and linker used to build an executable file
- Determine the relationship between a DLL function that is used and the overall purpose of the program
- Determine more details about unknown regions
- Identify the individual names of files stored in compressed file regions
- Detect the use of standard functions by way of function call signatures located in the code section

9. REFERENCES

- [1] @stake. SmartRisk Analyzer Press Release. @stake Inc. www.atstake.com. Accessed on August 22, 2004.
- [2] Cygwin. Cygwin Website. www.cygwin.com. Accessed on May 25, 2005.
- [3] Grimes, R. *Malicious Mobile Code*. O’Reilly and Associates, Sebastopol, CA, 2001, pp. 464-471.
- [4] HBGary. BugScan 2003 Press Release on July 31, 2003. HBGary Inc. www.hbgary.com. Accessed on July 22, 2004.
- [5] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format Specification Revision 6.0 February 1999. Microsoft Corporation. www.microsoft.com. Accessed on May 25, 2005.
- [6] Schaeffer, R. *Surfing Anonymously*. Data Becker, Newton, MA., 2002, pp. 96-97.
- [7] Tevis, J. *Automatic Detection of Software Security Vulnerabilities in Executable Program Files*. Dissertation submitted to the Department of Computer Science. Auburn University. Auburn, AL, 2005.
- [8] Tevis, J and Hamilton, A. “Methods for the Prevention, Detection, and Removal of Software Security Vulnerabilities”, *Proceedings of the ACM Southeast Conference*, Huntsville, AL, March 2004.
- [9] Viega, J. and McGraw, G. *Building Secure Software*. Addison-Wesley, Boston, MA, 2002, pp. 152-153.
- [10] Visual Studio. Microsoft Visual Studio Website. [Msdn.microsoft.com/visualC](http://msdn.microsoft.com/visualC). Accessed on May 25, 2005.