

REVERSE ENGINEERING VULNERABILITIES IN SIMULATION SOFTWARE

Eric S. Imsand, Adam C. Sachitano, and John A. Hamilton, Jr., Ph.D.

Department of Computer Science and Software Engineering

Auburn University

Auburn, Alabama 36849

E-mail: imsanes@eng.auburn.edu

sachiac@eng.auburn.edu

hamilton@eng.auburn.edu

Keywords: decompilation, reverse engineering, security, vulnerabilities

Abstract

As a result of the growing number of foreign powers that possess ballistic missile capability, the United States government has directed that a national missile defense system be constructed. Since its creation, the missile defense program has been one of the most sensitive programs in the United States government. Because of the continued sensitivity of this program, it became necessary to examine any and all objects associated with missile defense. Chief among those objects are the simulation packages used to model national missile defense.

The goal of this study was to determine the amount of sensitive information that could be derived from a simulation using only the compiled executables and the partial source code listing that is provided by the manufacturer. The analysis took place over approximately one month using a research team of two individuals. Focusing on both the Windows and Solaris releases of a simulation package, the analysis was conducted in four phases.

The first phase of the analysis consisted of attempting to find vulnerabilities in the software that might allow a third party to seize control of the executable. These vulnerabilities generally searched for the presence of unbounded buffers that might be exploited to generate a buffer overflow attack. Once a potentially unbounded buffer was found, the application was run via a debugger to verify that it could be used to take control of the executable.

The second phase of the study focused on the disassembly and decompilation of the simulation executable, and analysis of the generated code.

Particular attention was paid to the readability, validity, and volume of generated code.

The third phase consisted of analyzing the compiled executables in an attempt to document any information that could be obtained without any decompilation. The executables were analyzed without the use of a debugger or any other disassembly application with the aim of extracting potentially sensitive information in the form of compiled-in strings and debugging symbols.

The fourth and final phase of the analysis consisted of an examination of the available source code for the simulation software. This examination was primarily concerned with finding any weapons data that was hard-coded and the presence of any additional security vulnerabilities not discovered in the previous steps of the investigation.

Phase 1: Vulnerabilities

The first phase of the analysis consisted of testing the Windows version for any security vulnerabilities that may be present. The search primarily focused on any unchecked/unbounded buffers that might allow a third party to take control of the executable. It should be noted that the simulation designers maintain that buffer overflow attacks against the simulation do not pose a risk, since all sensitive information is supposedly stored in user supplied files, not hard coded into the executable.

The search for security vulnerabilities was conducted by attempting to force the application to respond to improper inputs. These improper inputs included inputting strings where the simulation software expected numbers, and vice versa. Most attempts at forcing the simulation to work with improper inputs failed, with the exception of one method. It was discovered that the use of the "Tab"

key when placed in a non-empty text box could generate a segmentation fault in the simulation software. Upon generating the segmentation fault, the currently executing code was examined via a debugger, more specifically the Microsoft Visual C++ 6.0 debugger. After analysis of the currently executing code, it was determined that the use of the “Tab” key could be used to generate a buffer overflow attack.

Other than the “Tab” key vulnerability, no other apparent vulnerabilities could be found that might allow an attacker to seize control of the executable. As previously stated, it is unclear what advantages could be afforded by taking control of the executable.

Phase 2: Disassembly & Decompilation

The second phase of the study attempted to recreate a form of the source code by attempting to extract it from the compiled executable. Because it is generally assumed that foreign parties would not have access to the source code for the simulation software, this was considered particularly important. Clearly the extraction of a form of source code from nothing more than the compiled executables must have an impact on the security classification of the model.

Reverse engineering is defined as “the process of analyzing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of abstraction [Van Deursen 2003].” The phrase “the extraction of a form of source code” clearly falls in line with this definition, though our search is primarily focused on finding sensitive data concerning missile defense. Generating a full and working source for a million plus line of code program is certainly not impossible.

Preliminary efforts focused primarily on the decompilation of the simulation source code. Decompilation is generally considered to be the generation of high-level source code from low-level input [Breuer 1994]. Unfortunately, the majority of the work that has been done on decompilation has taken place in private industry, meaning that much of the research that exists is proprietary [Housel 1974]. The goal of the decompilation experiment was to generate high level C or C++ source code

given the input of the compiled executables. If successful, this generated code would clearly be much easier to read than assembled code generated by disassemblers. In order to accomplish this goal given the tools that were available in the public domain, a key assumption had to be made. Most decompilers available for public use were designed to decompile C code. In other words, most currently available decompilers are designed for use on programs known to have been originally written in C or C++. Given the popularity of the C programming language, and the results from Phase 1, in which unbounded buffers were detected, this seemed like a reasonable assumption.

The first attempts at decompilation were made on the Windows platform, using several freely available utilities from the Internet. The first utility, known as “DCC” was designed to generate C code from a specified executable, without regard to the original compiler used to create the executable [Cifuentes 1994]. The second utility used, known as “DisC”, was also designed to generate C code, but was intended only for applications that were known to have been compiled using Borland’s Turbo C compiler [Kumar 2001]. The final utility, known as REC, was available for both the Windows and Solaris platform, and claimed to produce C code from compiled executables.

Neither DCC nor DisC successfully produced valid C code. The version of DCC that was used was too old to support the 32-bit executable structure used by the simulation. DisC failed for reasons that were never determined. REC ran for some time, but eventually crashed after producing about 28KB of output. Since the output generated by REC was incomplete, the correctness of the code it produced could not be determined. In other tests using smaller executables REC did run to completion, but did not appear to produce any valid code. None of the output generated by the Windows version of REC was ever successfully recompiled back into the executable.

Partial de-compilation of the Solaris version was achieved using the Solaris edition of REC. Testing of the Solaris version of REC on a trivial “Hello, World” example yielded unreadable and invalid C code. The “Hello, World” original C source file was 79 bytes in length. The REC de-compiled source was 8,034 bytes in length, including some auxiliary

information appended by REC. When this information was removed, the resulting source was still 4,664 bytes in length. As stated above, the code very hard to read, and did not compile back to the original executable.

It is thought that better, more specific, commercially available decompilers would perform better than REC on either platform, as REC is targeted to recognize several different object file formats targeted at many different processors. REC is also in an early stage of development.

Complete decompilation of an arbitrary program, while theoretically possible, is generally conceded to be economically infeasible because of the large number of instruction sequence combinations [Housel 1974]. Many decompilers are written using an ad-hoc method of the automatic decompilation of certain pre-classified instruction code sequences (for example, common arithmetic sequences), and handling certain other recurring sequences manually as “special cases” [Housel 1974]. In other words, it is hard to decompile an arbitrary program because of these special cases. However, if one is not trying to decompile programs in general, and is concentrating on a specific case (such as a package which may be suspected to contain sensitive information) a more focused attack may give better results. It should also be noted that code produced by optimizing compilers is considerably more difficult to decompile [Breuer 1994]. A side effect, therefore, of producing optimized code is that decompilation of an executable produced in this fashion will be more difficult to attack.

After failing to produce usable code via decompilation techniques, a complete disassembly of the model was attempted. A variety of publicly available disassemblers were tried. The majority of useful information was derived from two disassemblers in particular. Both packages were available for free on the shareware site download.com (<http://www.download.com>). The first package, known as PE Explorer came with a finite trial period, while the other package, known as Hackman, was free. Both packages were relatively small downloads, with Hackman being the larger of the two at roughly 3 megabytes. PE Explorer was tried first. PE Explorer provided valuable information concerning the simulation executables, such as the version number of the linker used to

create the program. Unfortunately, for reasons that were never discovered, PE Explorer was not able to successfully disassemble the simulation. After downloading, the Hackman software package installed quite easily. Aside from specifying the directory the application should be installed in, there were no questions that needed to be answered in order to install correctly. Hackman also requires that a newer version of the Visual Basic runtime libraries be installed, but these are available for free download from Microsoft.

The Hackman application also ran easily. After launching the program, it was simply a matter of selecting the tool, either a hex editor or disassembler, and choosing the file to open. Hackman opened the file, and disassembled it without further user interaction. The entire disassembly process took approximately six hours running on a 400 MHz Intel Celeron processor with 128 MB of RAM. After disassembly completed, the user has the opportunity to select a file to print to, for later reference. The disassembly of Hackman resulted in legal assembly code that was later re-assembled into an exact copy of the simulation executable. The assembly code totaled about 500MB in size, and was therefore difficult to work with. However, it should be noted that the resources available to foreign governments are considerably larger than those available to the researchers participating in this study.

Disassembly of the distributed executables in Solaris was achieved with “dis” on a Sun SPARC-based workstation running Solaris 8. Dis is a disassembler included in Solaris. Testing of dis on the same trivial “Hello, World” example described above yielded an output of 7,317 bytes. This includes a great deal of memory offsets in the output as well as the hexadecimal representation of the generated assembly. Because of this, it is not possible to re-assemble the direct output of “dis” command. However, a method of translating this output, along with the output of the “nm” command and the “dump” command can easily be found in the “comp.unix.solaris” usenet newsgroup.

All told, 440,168k of assembly code was generated from three executable files totaling 102,736k in size. This volume of generated assembly code represents approximately 9.3 million lines of assembly code.

A research paper found by the researchers discussing the reverse engineering of large legacy software systems concludes that the reverse engineering of such systems is 'intractable' in the sense that if one is given real [high-level] legacy code, the time required to show the validity of an explanation for why it exhibits a certain behavior is at least exponential compared to the size of the source code [Weide 1995]. However, the same paper asserts a caveat, which is repeated here: "This does not mean that the task is impossible. It means that it is prohibitively costly for large ... systems [Weide 1995]."

It should be noted that an organization with sufficient funds (read: a hostile foreign government) would almost certainly be able to compromise the simulation code we tested, even if given nothing more than the compiled executables. With sufficient manpower, the large amount of assembly code could be understood and mapped out. Also, given the number of viable decompilers that are targeted at specific compilation platforms, along with available theory on how to attempt such focused efforts [Housel 1974, Breuer 1994, Weide 1995], it would be possible for an organization to implement their own custom decompiler specifically tasked with compromising a single executable. Things that might thwart such an effort would be the use of optimizing (or other obfuscating) compilers and the continued secrecy of the compilation platform used by the original developers, among other possibilities discussed later.

Unfortunately, even in cases where the documentation does not explicitly state which compiler was originally used, it is possible to make an educated guess. For example, the software package PE Explorer stated that the version number of the linker used in the creation of the simulation was 6.0. Armed with that knowledge, a careful examination of commercially popular compilers can lead to discovery of the compiler used.

Phase 3: Analysis of Compiled Executables

Because of the overwhelming amount of assembly code generated by the disassembly of the simulation, it was decided to attempt to scan the compiled executables for any strings that might be left over from compilation. To accomplish this goal, a small program was written that attempted to parse

readable text out of binary files. This program took a compiled executable as input and parsed the file one byte at a time. Any two consecutive bytes that were readable English characters were then written to an output file, along with the rest of the line that contained the text. Using this method it became possible to parse readable text out of the compiled executable.

Not surprisingly, more information was obtained from the Solaris version than the Windows edition. The only readable text retrieved from the Windows version simply stated, "This program cannot be run in MS-DOS mode." Aside from that one statement, no other text strings were recovered from the Windows version.

The Solaris version yielded more interesting results. However, the results alone were not enough to derive any potentially sensitive information.

Approximately 27 megabytes of string literals were extracted from three executables. Just fewer than 1.6 million individually discernable strings greater than or equal to four characters in length were generated. Note that a large number of these strings are "trash" strings having no English-language meaning, or are object-file specific strings which have only partial English-language meaning, and which are used in computing the offsets of individual data members in certain aggregate data types.

Of the slightly less than 1.6 million strings literals found above, less than one-third, or about 450,000, of these were found in the initialized data space of the executable. This included what appeared to be function names and variable/member names. Between 32,000 and 36,000 of these string literals appeared to be format strings of the type used in standard I/O print statements. These included everything from error statements such as "Error, cannot open file %s for reading." or "MAJOR ERROR!!! [System/Ruleset/Sensor/Com Device/Jammer] %s does not exist for opfac %s," to other informational statements such as, "The following Platforms have the '%s' system type:" or "The following Systems use the '%s' system as a weapon ..."

In summary, about 1.6 million string literals were generated from three distributed executables. A large number of these were found to be either "trash," function calls, or variable/member names

(which can be more meaningfully researched in a source code analysis: see Phase 4). It is supposed that no meaningful information can be gleaned from these extracted string literals.

At the end of the analysis of the compiled executables, it was discovered that neither the Windows nor the Solaris versions had been stripped of debugging information. This is particularly disturbing given the information that could potentially be obtained simply by running the application through a debugger.

In order to evaluate the differences between the stripped and un-stripped executables, the Solaris version was run through the utility known as “strip.” Strip is included as a standard utility with Solaris. Stripping the executables and running a string literal search on the resulting output executables yields over 1.4 million string literals instead of approximately 1.6 million. The difference is about 166,000 string literals. The number of strings contained in just the initialized data spaces of the executables remains the same, as no symbol debugging information is contained in these areas.

The result was that the stripped executables were somewhat more difficult to analyze with a debugger. Instead of being able to view the mnemonic variable and function names during execution with a debugger, variables were referenced by address, or a meaningless, generated name. For example, if someone wished to run the executable in a debugger, they could set a breakpoint at the main() function using the non-stripped version of the executable. After the executable had been stripped, it became necessary to know the exact memory offset of the main() function in order to set a breakpoint. In short, analyzing the flow of such a program would be an extremely difficult task.

It should be noted that, simply because the executables have not been stripped, does not mean that the source code can be viewed using a debugger. The presence of the variable and function names simply makes it easier to analyze the flow and function of the program.

Phase 4: Source Code Analysis

The final step in the analysis of the simulation was the examination of the model source code included in the distribution. Source code for the model is an important element in

effectively using the simulation. We limit our discussion in this forum merely by noting that only extensive parameterization of sensitive information makes sharing the source code of sensitive models.

Conclusions and Recommendations

After completion of a vulnerability analysis of the simulation executables and the accompanying model source code, several conclusions were reached. These conclusions must be considered with the knowledge that the researchers participating in this study have no experience or expertise in missile defense, nor did they have access to classified materials on which to base their conclusions.

The first conclusion drawn was that there are apparently vulnerabilities in the code that would allow a rogue piece of software to seize control of the executable. This attack would more than likely come in the form of a buffer overflow attack, or some similar method. It is unclear what advantage taking control of the executable would provide, though it is possible that there may be an unknown reason why this would be advantageous for an organization seeking access to sensitive information.

Second, there does appear to be a large number of string literals present in the compiled executables on the Solaris platform. Most of these string literals did not appear to be anything remotely sensitive in nature, although many of them appeared to be function and variable names.

Third, it appears that neither the Solaris nor the Windows version of the simulation had the debugging information stripped from the executables at compile time. This provides anyone insight into both the names and purposes of the functions of the simulation. This information was quite easy to retrieve – the only necessary tool was a debugger.

Fourth, there are quite a few instances of potentially sensitive information still in the source code distributed to all US users of the simulation. These instances were usually found in comments still placed in the code, usually detailing upgrades that were made to the software. There were few instances of values being hard-coded into the source code. Most of the values appear to be input from another location, such as a file or the keyboard, or declared in header files that were not included with the simulation distribution.

Based on the preceding conclusions, it is recommended that, at the very least, the executables be recompiled in order to strip debugging information from them. There does not appear to be a disadvantage to performing this action. In addition to recompilation, it is recommended that the simulation source code be reexamined in order to determine the sensitivity of the contents. A better solution might be to discontinue the practice of distributing the source code for the models altogether.

References

- Breuer, Peter T., Bowen, Jonathan P. 1994. Decompilation: The Enumeration of Types and Grammars. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16 no. 5.
- Cifuentes, Cristina. 2002. The dcc Decompiler. Website - <http://www.itee.uq.edu.au/~cristina/dcc.html>
- Housel, Baron C., Halstead, Maurice H. 1974. A Methodology for Machine Language Decompilation. In the proceedings of the 1974 ACM/CSC-ER annual conference. ACM, New York, NY. 254-260.
- Kumar, Satish. 2001. DisC – Decompiler for TurboC. Website - <http://www.debugmode.com/dcompile/disc.htm>
- Muller, Hausi A., et. al. 2000. Reverse Engineering: A Roadmap. Proceedings of the Conference on The Future of Software Engineering (Limerick, Ireland). ACM, New York, NY. 47-60.
- Rajala, Norman, Campara, Djenana, Mansurov, Nikolai. 1999. Insight: Reverse Engineer Case Tool. Proceedings of the 21st International Conference on Software Engineering (Los Angeles, California). ACM, New York, NY. 630-633
- Van Deursen, Arie. 2003. Reverse Engineering. Website - <http://www.program-transformation.org/twiki/bin/view/Transform/ReverseEngineering>

Weide, Bruce W., Heym, Wayne D., Hollingsworth, Joseph E. 1995. Reverse Engineering of Legacy Code Exposed. Proceedings of the 17th International Conference on Software Engineering (Seattle, Washington). ACM, New York, NY, 327-331

About the authors

Eric S. Imsand is a 2002 graduate of Auburn University, where he earned a Bachelor of Science degree in Computer Science. He is currently pursuing a Masters degree in Software Engineering at Auburn University, with an expected graduation of December 2003. His research interests include anti-virus technologies, network and operating system vulnerabilities, NP complete problems, and database theory. He is a native of Huntsville, Alabama, and currently resides in Auburn, Alabama, where is employed as an instructor and researcher for Auburn University.

Adam C. Sachitano is currently an undergraduate student pursuing a Bachelor of Science degree in Computer Science at Auburn University with a non-major concentration in Applied Discrete Mathematics. He expects to graduate in December 2002 and remain at Auburn to pursue the degree of Master of Science in Computer Science. His research interests include programming languages, simulation, security, algorithm analysis, and embedded software development. He is a native of New Orleans, Louisiana, and currently resides in Auburn, Alabama, where he is employed as a software developer for a local company.

John A. “Drew” Hamilton, Jr., Ph.D., is an associate professor of computer science and software engineering at Auburn University. He has a B.A. in Journalism from Texas Tech University, an M.S. in Systems Management from the University of Southern California, an M.S. in Computer Science from Vanderbilt University and a Ph.D. in Computer Science from Texas A&M University. Prior to his retirement from the US Army, he served as the first Director of the Joint Forces Program Office and on the Staff and Faculty of the United States Military Academy. CRC Press publishes his book, *Distributed Simulation*, written with LTC David A. Nash and Dr. U. W. Pooch.