

# VULNERABILITY OF SIMULATION EXECUTABLES

**Brian Eoff and John A. Hamilton, Jr., Ph.D.**  
**Department of Computer Science and Software Engineering**  
**Auburn University**  
**Auburn, Alabama 36849**  
**E-mail: [oeffbri@auburn.edu](mailto:oeffbri@auburn.edu), [hamilton@eng.auburn.edu](mailto:hamilton@eng.auburn.edu)**

**Keywords:** Security, Modeling, Simulation, Reverse Engineering, Decompilation

## ABSTRACT

The creation of large-scale simulations is becoming more prevalent. Simulations are used in wide ranging fields, from network design to military simulation to automobile design. These simulations allow a company to test products, and do quick design changes. To change the rear bumper angle on a Ford Excursion in a simulation is as simple as changing a variable, and then letting the simulation run a real world test that could compute how such a change might affect gas mileage. Doing the same experiment in the real world would not be cost effective. A physical prototype would have to be created, and the product would have to be secretly tested, outside the always-watchful eyes of competitors. Also this simulation could be used across the whole product line.

Simulations such as these are expensive. Trade secrets must be encoded into the simulation for it to work accurately. Ford would have to include certain facts that it considers known for a simulation work properly. The security question arrives from the possibility that a competitor might gain access to these facts. Chrysler could gain a wealth of information from merely knowing the parameters Ford uses to test its products.

Companies need to protect their executables. With the introduction of more advanced decompilers a competitor can use the executable to withdraw information. This is not limited to simulation. Any expensive software where trade secrets are stored in is vulnerable to such an attack. High-end graphics and animation programs could also be a target. By decompiling an animation program a competitor could learn what assumptions the program makes when it animates hair, or some other mathematically complex

operation. This isn't a concern in other software development. The reward of decompiling Microsoft Word is minimal; it is a word processor, and does nothing ground breaking. A competitor has nothing to gain by learning how Word does spell checking. With highly specific simulators though, a competitor can gain a sizable amount of information about its competitor.

In this paper we will discuss the possible solutions to protecting a simulation executable. We will not limit my discussion to technological discussion; we will also talk about the possible legal ramifications of such industrial espionage.

In the end it comes down to this, reverse engineering a simulation is rewarding enough to invest the lengthy time needed to do so. In many cases a complete reverse engineering of the simulation is not necessary, simply getting the important values out of the executable is sufficient enough. This can't be said for many other commercially available pieces of software.

## 1 INTRODUCTION

“Remember that hiding secrets is hard.” – Gary McGraw

Simulations have the same problem that all expensive software has. Once the developer is finished it is out of their hands. They cannot take the software back when a user begins to misuse the program. For most software development this is not a problem. The time it takes to discover the secrets of the executable versus the information gained is extremely one sided. What a developer needs to protect are the trade secrets in their software. These secrets could be algorithms or numerical assumptions that the software makes. One commercial example is high-end graphics and animation programs. They contain many closely guarded algorithms that if

obtained by a competitor could cause them to gain an advantage in the market place.

Simulations raise a greater concern. They are large pieces of expensive software that are algorithmic intensive and contain many numerical assumptions. In a networking simulation, for example, how the clock is defined is a massive problem. A competitor could find out how the clock is used, and simply take the idea.

A greater concern to national security is at stake though. The biggest customer of simulations is the United States Department of Defense [Balci 2001]. These simulations are used to plan battles, and predict the outcome of military actions. In these simulations are numerous secrets. Some secrets could be weapon ranges and potential damage caused by weapons. The United States shares these simulations with allies, but most likely does not want all this information known.

This paper will cover how a malicious user can gain access to the internals of a simulation, by using a variety of techniques on the executable. These techniques include decompilation, disassembly and monitoring the behavior of the simulation. Also included is a small section on buffer overflows. The techniques that developers can use to protect their simulation is also presented; they include obfuscation, encryption and the use of a client/server model. Finally there is a discussion of how developers can use the techniques incorporated by virus writers to protect their executables.

## **2 COMPROMISING INFORMATION**

Once a malicious user has the executable in their possession they have numerous ways to garner information. Some of these techniques are easier than others, decompiling an executable and looking at the source is by far easier than entering in hundreds of thousands of inputs and trying to make assumptions of how the program behaves by recording the output.

### **2.1 Disassemblers**

A disassembler is a program that takes in machine code and outputs equivalent assembly language. Often disassemblers come packaged with debuggers. When a program is converted to assembly by the disassembler looping structures are converted to counters and jump statements, which are not as easy to understand as the original high-level language code. Despite this, mathematical equations can be derived and string literals are still present. A string literal is

the outputted string text, an example is `printf("My name is");` in C. Once a program with this statement is assembled and compiled, and then a disassembler is run on the program, the string "My name is" can be found. The only information lost is the "printf" function name.

To get useful information out of a disassembler requires a basic understanding of machine language. Once that is known though any mathematical equation used or any defined numbers can be obtained. The basic functionality of the program can also be determined.

### **2.2 Decompilers**

Decompilers turn machine code directly into high-level languages such as C or Java. Decompilers are not as mature a technology as disassemblers. The output of a decompiler is much easier to understand than the output of a disassembler [Cifuentes and Gough 1995].

Decompiling Java code is extremely simple because it is targeted for high-level machines. Java programs can be brought back to something very similar to source code with a decompiler. C programs though, do not often look similar to their original programs, because much information tends to get thrown away during the compiling process. Also if the debugging option is on during a C program compilation a decompiler has a better chance of reproducing accurate source code.

Some of the decompilers available are Mocha, OEW and Decaf for Java and REC and DCC for x86 to C decompilation

Many people have ethical questions about the creation of decompilers. Decompilers exist for more than just removing trade secrets from program. Decompilers are used by security specialist to track down flaws in commercial software. It would be a mistake for decompilers to be deemed illegal, which is a possibility with the DMCA. Is a binary executable an encrypted form of media, and do decompilers break that encryption? This is part of a dangerous trend of outlawing tools and not the crimes committed with them. Reverse engineering with decompilers, using them to steal trade secrets is illegal. The decompiler is nothing more than a tool.

### **2.3 Monitoring Behavior**

Monitoring the behavior of a simulation requires more

patience than decompiling or disassembling the executable, but it has been known to produce useful results. The encryption used to store Netscape Communicator passwords was broken this way. McGraw simply entered in passwords and watched how the program stored them. He documented the passwords and the resulting output. He studied the output and was able to determine the way Netscape encrypted the passwords. The same idea could be used on simulations.

A user could record the input he fed into the program and the subsequent output. Most likely there are multiple fields of input and multiple outputs. This is similar to black box testing; the tester can only view the input and output of a program. This technique is capable of finding many flaws in software. With very complex systems, such as simulations, monitoring behavior becomes very difficult. The simulation has multiple input values and output values, for a reverse engineer to determine how they correspond would be a near impossible task. It would require a large amount of information and the ability to search for patterns and connections. For smaller program with few inputs this technique is reasonable, for a simulation it is near impossible. The only chance is to limit the program into sections where a few inputs make an output, and attempt to dissect the simulation section by section.

## **2.4 Buffer Overflows**

Buffer overflows are a widespread problem in software development. The problem is simple; programming languages such as C does not do adequate bounds checking. It allows a user to input more data than the buffer can handle. Many times this security flaw is used by a malicious to allow an executable to be run with more privilege than it would otherwise have. A buffer overflow allows a malicious user to change the return address of a function. This can cause the flow execution of the program to be changed. Buffer overflows are a concern in simulations because simulations often take in many data inputs. A simulation written in Java, while not recommended, would not be victim to buffer overflows due to the built in bounds checking.

## **3 HOW TO PROTECT THE SIMULATION**

“If we cannot make reverse engineering impossible, we can at least make the task costly in terms of time and effort” – Douglas Law

Simulation creators should be concerned, but they should not feel helpless. There are ways to protect the sensitive information in an executable. Also a simulation is an extremely large and complicated program. It is quite a task to gain an understanding of its entire structure and secrets. This is the same reason people don't decompile the Windows OS, it is too massive to fully understand. A focused malevolent though may need to only gain a few secrets to consider himself a success though.

### **3.1 Good Coding Standards**

The choice of programming language, compiler, and design of the program are critical for protection of the simulation. An interpreted language such as Java is a bad idea for code protection simply because it is such a trivial task to decompile Java class files. Also setting the right parameters on a compiler, turning debugging options off will make it harder to decompile the executable. Also not including information that user should be able to input, such as data, is a good idea.

This doesn't fall on the programmers, but the distribution of the simulation software should be monitored. This is especially important in the defense industry. If in doubt a “weaker” version of the simulation should be released to parties that developers are unsure of. Weaker meaning limited capabilities and data sets.

### **3.2 Code Obfuscation**

Security by obscurity is an idea looked down upon by almost all security and cryptology experts, but it is a good start in protecting executables. The obscuring technique is called code obfuscation. Code obfuscation is changing the program code in such a way that it becomes more difficult for attackers to read and understand, yet it functionality identical to the original. The program must produce the same results, but it may execute slower, or have additional side effects due to added code [Law 1998]. There are many different types of obfuscation techniques. Below are three of the most common methods.

#### **3.2.1 Layout Transformations**

Layout obfuscation is when information that is not necessary to the execution of the program, such as variable names and comments, is altered. This is also commonly referred to as lexical transformations.

Typically all this entails is scrambling identifiers names. This will prevent some thievery, but any determined reverse engineer will be able to read past the scrambling of identifiers to see what the code is really doing [Collberg 2002]. There are programs available that will do layout obfuscation; the C Shroud system is a layout obfuscator for the C programming language [Law 1998]. Java has a similar layout obfuscator Crema [Collberg et al.1997]. Layout obfuscation is a good first step, identifiers contain a good deal of practical information, but it not enough by itself. Layout transformation is the simplest form of obfuscation.

### 3.2.2 Data Transformation

The goal of data transformations is to obscure the data structures used in the source application. These transformations can be classified as affecting storage, encoding, aggregation or ordering of data. Obfuscating storage transformations attempt to choose unnatural storage classes for dynamic as well as static data. An example is Boolean values can be split into two or more variables; this is due to the limited range of Boolean values. Encoding transformations chooses unnatural encodings for common data types. Aggregation is used to hide a programs data structures. This includes splitting arrays or merging multiple arrays into one array. Ordering transformation is used to randomize the order of declarations in a program [Collberg et al. 1997].

### 3.2.3 Control Transformations

Control transformations can be classified as affecting the aggregation, ordering or computations of the flow of control. Control aggregation transformation breaks up computations that logically belong together or merge computations that do not. Control ordering transformation alters the order in which computations are carried out. Computational transformations insert redundant or dead code into the program. For obfuscations that alter control a certain amount of computational overhead will be unavoidable. For a developer this means having to chose between a highly efficient program and a highly obfuscated program. The biggest issue with control transformation is making them computational cheap, yet hard to deobfuscate. The problem is similar to that of public key encryption where it is relatively easy to

check if a number is a factor of another number, but it is hard to factor a large number [Collberg et al. 1997].

### 3.2.4 Programs That Obfuscate

There are numerous programs available that do code obfuscation. Some of them are quite simple and do nothing more than layout transformation, which will provide little security. Other more advanced programs are able to do data and control transformations.

### 3.2.5 Encryption

Encryption of sensitive areas of code is another possibility. A developer could simply encrypt algorithm and data, and upon a legal execution (i.e. not a disassembler or a decompiler) the sensitive areas would be decrypted and the program would function fully. Upon completion of execution the sensitive areas would be re-encrypted. Even if a developer used a good system of encryption, which is not always the case, this possible solution still has many flaws. The developer would just inherit the main issue with encryption, how to distribute the keys. If the key is stored in the software than a reverse engineer could simply get the key from the executable, and decrypt the sensitive data. The only solution would be for the encryption/ decryption process to take place in hardware, but that is not very feasible [Law 1998]. The key needed to decrypt could be stored on a server, and once the simulation began a request would be made for the key. As long as the communication is secure, this could potentially work. Also it would allow the developer to encrypt the data again with a new key. Every so often when the simulation makes a request for the key, it gets a command to after completion encrypt the sensitive areas with a new key. This would make the possibility for a brute force decryption attack even more unreasonable.

### 3.3 Client Server Model

A possible way to prevent reverse engineering of source code is to prevent physical access to the executable. A user would communicate with the program via an interface [Law 1998]. This idea has many negative issues in the simulation world. Simulations are too complex and computationally heavy to be run using the client/server model; latency and bandwidth become major issues. Using the same idea though, sensitive parts of the program can be kept

on a server and have the users machine run the rest locally. This still has many problems; the possibility exists that sensitive data could be still gotten by simply monitoring the communication channel that the program uses. Also there is the issue of how the client would authenticate itself to the server. A key system could be used, but that key would have to be stored locally, and thus be obtainable by a reverse engineer. Total server-side execution is the only known way to completely protect the executable from reverse engineering, but due to the sensitive nature of many simulations, purchasers would be wary to communicate over public networks. How to ensure communication is secure and unmonitored now becomes the main problem. One headache is exchanged for another.

### 3.5 Attacking Debuggers/Disassemblers

Another way for a developer to protect their code is to insert instructions that make disassemblers and decompilers fail. Cohen describes a way to make disassemblers fail on the Intel x86 series, "... we can include jump instructions whose last byte (usually the address jumped to) corresponds to a desired operation code and places the code jumped to appropriately so that the jump works and the proper return address is in the middle of the previously executed instruction. In this way, we reuse the last bytes of the jump location as operation codes on the next pass through the code" [Collberg and Thomborson 2002].

HoseMocha is an application that causes the Mocha decompiler to fail. HoseMocha appends extra instructions after a return instruction. The addition in no way affects performance, but it causes the Mocha decompiler to crash.

A security conscious developer should find out what decompilers and disassemblers could be used to get information from their program. Then they should attack flaws in these decompilers, by inserting whatever is necessary into their simulation to cause decompilers to fail.

### 3.6 Watermarking

Watermarking embeds copyright information into the program code that would allow the creator to assert his ownership. Watermarking is more of a legal precaution. There are many issues associated with watermarking; how to make sure the watermark can be reliably located and extracted from the program, that

the watermark does not adversely affect the performance of the program, and that the watermark is stealthy [Collberg and Thomborson 2002]. The watermark needs to contain some internal structure that will allow us to detect tampering. Parity or error-correcting bits may be used for this purpose.

Watermarking allows developers to insure ownership, but it will not keep a reverse engineer from getting trade secrets from the executable

### 3.7 Tamper-Proofing

Tamper proofing causes a program to malfunction when it detects it has been modified. Tamper proofing code can examine the executable program itself to see if it is identical to the original one. The malfunctioning mechanism could simply be inputting commands that would cause the program to exit on execution [Collberg and Thomborson 2002]. Tamper proofing prevents a malicious user from altering the programs, such as removing copy protection or registration information, but it does not keep a reverse engineer from looking at the code. Tamper proofing code should be installed, but it cannot be depended on to keep the programs secrets safe.

### 3.8 Keeping Trade Secrets – Remove Them

While this does not apply to algorithms, in many simulations certain values are predefined in the software. A safer way is to allow the user to input these values. An example could be a military simulation where the range of a weapon might be numerically defined in the software. The user should be prompted for this value; it shouldn't be stored in the executable. This might be viewed as an inconvenience to the user, but is a sure way of keeping secret numerical data secure. If the user is privy to the information, then entering it in should be feasible.

## 4 WHAT WE CAN LEARN FROM VIRUS WRITERS

Virus writers are notorious for hiding what their executable code does. They use techniques such as polymorphism and encryption to keep the internals of their executable code. Legitimate developers could learn much from the struggle between virus writers and the creators of virus detection software [Collberg and Thomborson 2002].

The polymorphic quality of some viruses should be given special attention. A polymorphic virus is able to change its basic structure upon each execution. A well-designed program could be able to use the same idea and obfuscate on each execution. This technique would make it hard for a group of reverse engineers to reconstruct a simulation; each one would have a differently structured version. This technique might interfere with tamper proofing techniques. Developers should decide which technique gives them the best security.

## 5 CONCLUSION

“Given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application.” – Christian Collberg

Binary executables are not secure, and any information in a developer’s code that he wants to keep secure could be compromised. This should be a major concern in the simulation field where trade secrets and military data are embedded into source code. Also due to the high cost of developing simulations a competitor could gain an unfair advantage by skipping the high cost of development and stealing ideas. Trying to prove that trade secrets were stolen would be very difficult, and legal resolution would be expensive. There are ways to protect the secrecy of the code contained in the executable. Like all security issues though the developers must weigh the cost and hassle of putting in devices to protect their code against the ease of using and maintaining their software. In the future with the rise of network connectivity we believe a server client model with strong encryption authentication will be used to transfer sensitive information. Until then the mere size and complexity of modern simulations should make complete and accurate reverse engineering difficult. Add to this obfuscation and it becomes even more overwhelming. A malicious user can view sensitive information in an executable, but using basic techniques, he will most likely not be able to understand what he sees.

## REFERENCES

Balci, Osman (2001), "[A Methodology for Certification of Modeling and Simulation Applications](#)," *ACM Transactions on Modeling and Computer Simulation*, Vol. 11, No. 4 (Oct.), 352-377.

C Cifuentes and KJ Gough, [Decompilation of Binary Programs](#), Software - Practice & Experience. Vol 25 (7), July 1995, 811-829.

C Cifuentes, T Waddington and M Van Emmerik, [Computer Security Analysis through Decompilation and High-Level Debugging](#). Proceedings of the Working Conference on Reverse Engineering, Workshop on Decompilation Techniques, Stuttgart, Germany, 3 Oct 2001, IEEE Press, pp 375-380.

Colberg, Christian and Thomborson, Clark. 2002. Watermarking, Tamper-Proofing and Obfuscation – Tools for Software Protection. *IEEE Transactions on Software Engineering*. Vol. 28 no. 8 (Aug) : 735-46

Collberg, Christian, Clark Thomborson and Douglas Low. Taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July, 1997.

Low, Douglas. 1998 Protecting Java Code via Code Obfuscation. *ACM Crossroads*. Vol. 4 no.3 (Spring)

## BIOGRAPHY

**Brian Eoff** is a senior at Auburn University majoring in Software Engineering with a minor in English. His area of interest is computer and network security, decompilation and Warhol Worms. He is the president of Auburn’s chapter of the ACM, and a member of the Auburn Water Polo team. Brian Eoff is 22 years old and currently lives in Auburn, Alabama. His hobbies include swimming, traveling, hiking, writing fiction, and sleeping. He plans to graduate Spring 2003, and will pursue graduate school in the subsequent fall. Brian was born and raised in Huntsville, Alabama, and has worked for Adtran and TRW.

**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 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.