Chapter 9

Security Issues Resulting from Interoperability

By
Dr. John A. Hamilton, Jr., Auburn University
Wade Chatham, Auburn University
Brian Eoff, Auburn University
Eric Imsand, Auburn University
Adam Sachitano, Auburn University

The views expressed in this chapter are the opinions of the authors, and do not reflect the official opinions of any U.S. government agency.

Once coalition networks are established, the vulnerability of information systems may increase. Internal propagation of a worm with the characteristics of, for example, “Nimbda” or “Code Red” can generate internal broadcast storms behind the network firewalls. There are significant limitations to applying simulation information security issues. Most security vulnerabilities occur at the end points and evaluating the actual systems best assesses these vulnerabilities. However, network simulation is an obvious choice for evaluating the impact of primary and secondary packet storms on large, internal networks.

This chapter will outline issues associated with communication system interoperability, the security vulnerabilities associated with improved interoperability, network simulation to support the evaluation of the effects of worm-induced packet storms and the experimental design, implementation and results using the OPNET simulation environment. Additionally, this chapter will present sanitized research results of security issues associated with distributed military systems and the vulnerabilities associated with shared software.
9.1 INTEROPERABILITY

Communication system interoperability, the ability of two or more systems or components to exchange data and use information [IEEE 1990], is inherently software-based. Interoperability implies the existence of diverse systems that need to exchange data and services. Much is written about “systems of systems.” System interoperability is what makes heterogeneous systems of systems a reality. All of these systems are composed of hardware and software. Hardware is not easily changed. Furthermore, fielded hardware systems often cannot be wholly replaced.

Diverse hardware-based communications systems require an overall software architecture in order to interoperate. As noted in IEEE Standard 12207.0-1996 Software Lifecycle Processes, software architecture describes the top-level structure of the over-arching system and describes the software components. Specifically, developers adhering to the standard are required to develop and document a top-level design for the interfaces external to the software item and between the software components of the software item. This is an essential first step in achieving interoperability between any two systems.

In Defense applications, interoperability is seriously hindered by the sheer number of systems, standards, and system developers / procurers. Figure 1 illustrates some, though no means all, of the commands developing/fielding software-intensive communications systems.

![Figure 9-1. Some organizations involved in command system acquisition: DISA, NSA, three service C2 acquisition commands and ten unified commands.](image)

An exemplar “system of systems” is the Global Command and Control System (GCCS) as shown below in Figure 2. A key factor in the success of GCCS is the discipline to which interface standards are maintained by the Defense Information Systems Agency. GCCS is a non-trivial system with at least twenty-three systems.
The purpose of this chapter is not to discuss the relative merits of the design and implementation of GCCS, but merely to illustrate the very complicated nature of military IT systems. Software makes the vision of “seamless interoperability”
possible. But software strategies are often thwarted by scale and complexity. GCCS is just one example of multiple systems from multiple organizations. Interoperability, particularly data interoperability, between systems is achieved primarily through software.

Consider synchronizing a handheld with a desktop as illustrated in Figure 3. There are potential connectivity issues, whether USB, Serial or infrared ports are used. Fortunately for consumers, these ports are well defined. Connecting the ports is not enough. There must be compatible software on both the desktop and the handheld to exchange data over the connection.

DOD interoperability issues are so challenging that it is easy to overlook the security risks that come with interoperability. Consider the following remarks from Admiral Dennis C. Blair, USN regarding GCCS and interoperability [Blair 2001]:

“There are also GCCS incompatibilities in combined operations; for example, GCCS Joint and GCCS-Korea. These two systems share some common operational picture data, but do not share information via files, e-mail, and other web service tools. Obstacles to combined interoperability lie in information release restrictions. Our allies understandably restrict release of their classified information. Likewise, we want to control release of U.S. classified information. To achieve effective combined interoperability, we must develop much more capable security procedures and sophisticated tools to allow information exchange while protecting our national and allied data.”

9.2 INTEROPERABILITY & SECURITY RISK

As Admiral Blair noted, in the previous GCCS example, the interoperability issue was a policy issue, not a technical issue. Policy and technology together can be potent challenges. Consider a requirement US Pacific Command articulated needing a secure email system to exchange sensitive but unclassified information between Headquarters, Australian Theater and Headquarters, US Pacific Command as illustrated in Figure 4.

Figure 9-4 Secure email in the Pacific
The alternatives to secure emails were frequent, incredibly long facsimile messages exchanged over secure telephone lines.

Unfortunately, even limiting data exchange to email applications increases risk to all stations enabled for interoperability. Viruses can spread through email so the more stations connected, the greater the risk for attack. Worms, malicious programs specifically designed to replicate across networks are continuing threat.

The Morris Worm, which attacked networked stations in October 1988, was extensively documented [Spafford 1989] and [Rochlis & Eichin 1989]. Although a well-documented phenomena, network vulnerability to worm attacks has only increased. New worms continue to proliferate and virus scanners are updated after the initial attacks. Antivirus responses while eventually effective are purely defensive measures.

Table 9-1  Lifecycle of a typical Nimda Worm variant

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>Local executable file infection (prepending)</td>
</tr>
<tr>
<td>Stage 2</td>
<td>Search for IP addresses from registry</td>
</tr>
<tr>
<td>Stage 3</td>
<td>Elevate privilege and execute on vulnerable</td>
</tr>
<tr>
<td></td>
<td>remote computers</td>
</tr>
<tr>
<td>Stage 4</td>
<td>Compromise local security settings</td>
</tr>
</tbody>
</table>

Thirteen years later, the Nimda Worm demonstrated the vulnerability of modern email systems and webservers. The Nimda Worm was identified on 11 October 2001 and was found to have a four-stage lifecycle as shown in Table 1.

Consequently, it is straightforward to plan and execute a timed distributed denial of service attack (DDoS). Significant effects have been achieved through ad hoc, and sometimes amateurish attacks. [King, Dalton & Osmanoglu 2001] note that there are many variants of denial of service attacks such as:

- Programming mistakes that take 100% of CPU time.
- System memory usage may continually increase due to a memory leak.
- Malformed data requests such as Web requests or remote procedure calls (RPCs).
- Large packets such as email addresses and Internet Control Message Protocol (ICMP) requests.
- Non-stop network traffic User Datagram Protocol (UDP) and ICMP (broadcast storms and network flooding.
- Forging routing information or unresponsive connection requests.
- Incorrectly configured wiring, power, router, platform, or application.

The authors further note that the Computer Emergency Response Team (CERT) has documented more than 318 DoS attacks.
A distributed DoS is merely the employment of a distributed set of remote computers to intensify the impact of the attack. An DDoS attack launched in conjunction with an email-propagated worm with a specific military objective could successfully “jam” military computer networks at a critical time of the attacker’s choosing.

Consider the following attack scenario:

- Select desired date/time for denial of service attack initiation in worm payload.
- Study resources on the internet to assist in the design of the worm.
- Use lessons learned from various white hat and black hat security sites to maximize surprise elements of worm.
- Use “human engineering” to ensure that worm is introduced undetected on as many target machines as possible.
- Reinforce DoS attacks remotely through any backdoors opened by the worm at h-hour.

Even if the target network is only affected for a few hours, it may be a militarily significant impact. It is a reasonable precaution to evaluate network vulnerabilities to DoS/DDoS from internal as well as external stations.

9.3 NETWORK SIMULATION AND DDOS VULNERABILITIES

A research group at Auburn University undertook a serious analysis of one network on campus. After modeling the network in OPNET, one pair of students studied the predicted impact of increased multimedia traffic on the network while the other pair focused on evaluating the impact of DoS attacks on the network.

OPNET may be described as a communications-oriented simulation language. The name OPNET is derived from Optimized Network Engineering Tools. The single most significant aspect of OPNET is that it provides direct access to the source code coupled with an easy-to-use front end. This capability allows the introduction of multiple traffic sources, from PDFs, from network emulators and from observed traffic.

OPNET uses the following modeling hierarchy as shown in Figure 5 [Hamilton, Nash & Pooch 1997]. Node models were used to represent the CISCO 5500 switches found in Auburn’s Broun Hall as shown in Figure 6.
Broun Hall has four layer three switches that handle all of the traffic between the internal computers. Three of these connect to a main switch via gigabit fiber optic cable. The main switch connects to the main campus switches via gigabit fiber optics as well.

The students modeled the switches as follows: Each wiring closet was represented by a single switch, each LAN by a single node, and RoTW represented the source/destination of all inbound/outbound traffic respectively.
Verification of a simulation is the process of assessing the degree to which the implementation transforms inputs into outputs as specified by the model. The ultimate verification test is to model a known system and run the same sets of inputs through the actual system and the simulation. If the results are statistically the same, then you have reasonable assurance that you have implemented the model correctly.

This was the approach used by the students using data collected by the College of Engineering. In the course of the experiment, the students learned first-hand that network monitoring is a non-trivial effort. In trying to verify the model, it was subsequently determined that the actual time scale of the network monitoring system was not properly understood by the system administrator. The students successfully verified that their model accurately represented the actual network for a given load.

Validation is the process that establishes the extent to which a model does (or does not) acceptably represent the phenomenon of interest. Once we know we have a good model, how do we gauge its predictive power? Generally a good traffic model is necessary for a network simulation to achieve any meaningful predictive power. Network traffic patterns cannot be relied upon to follow a standard probability distribution. The failure of Poisson processes to model network traffic is outlined in [Paxson & Floyd 1995].

Accurate modeling of network traffic can limit many network simulation studies, but is not an issue in modeling DoS attacks. For a given worm, it is possible to exactly replicate the initiation time and traffic volume of each infected node. Network simulation is particularly appropriate for studying large-scale distributed DoS attacks since it is usually too costly to use large distributed systems for live DoS/DDoS testing.

Both groups wanted to see the effects of increased traffic on the network topology. They used the built-in traffic scaling functionality of OPNET to scale traffic with 1000% and 5000% increases [D’Amico & Taylor 2002]. Increasing traffic beyond 5000% overwhelmed the network.

The network selected to study was very lightly loaded, usually averaging traffic of 3.5 megabits/sec percent loaded based on the data collected by the university network staff. The students observed that the Cisco switches had greater capacity than the gigabit links. The Cisco white paper regarding the Cisco 5500 switch, rates the switch at 50 Gbps maximum throughput. The technical specifications of those switches were found below at:
It was clear that there were simply not enough gigabit links feeding into the
switches to overload the switches, thus the throughput on the links was the
theoretical throughput limit.

The teams reported the following results: Figure 7 shows that the throughput
(bits/sec) maxes out at 1 Gbps.

![Throughput limited to 1 Gbps](image)

**Figure 9-7 Throughput limited to 1 Gbps**

This is expected because the gigabit link can have no higher capacity than its rated
capacity of 1 Gbps. Consequently, in Figure 8 the group observed that utilization
behaves in direct proportion to maximum capacity.

![Utilization behaves in direct proportion to max capacity](image)

**Figure 9-8 Utilization behaves in direct proportion to max capacity**

Based on this study several conclusions were drawn regarding the vulnerability of
the network to an internal DoS/DDoS attack. First it was observed that the
network was generally very lightly loaded. Second it was observed that each
switch had significantly greater processing capacity than a single gigabit
connection. Given the current topology (to include the limited number of stations
on the network), it would be very unlikely to that an internal DoS attack could
overwhelm a switch. However individual link failures could be expected.

System interoperability is a force multiplier for command and control systems, but
added capability increases risk. More connected stations increases the risk.
Assessing the vulnerability of a network to DDoS attacks is prudent. This work demonstrated one practical means to evaluate such vulnerabilities using OPNET. Although simulation can be used to study denial of service attacks, simulations themselves can be a source of vulnerability. In the next section, Brian Eoff describes the vulnerability of simulations themselves. As distributed simulations are shared with coalition partners and used for training, the price of increased interoperability is increased vulnerability. This chapter will conclude with a sanitized analysis of a US military simulation released to support interoperable coalition battle management command and control (BMC2).

9.4 VULNERABILITY OF SIMULATION EXECUTABLES

"Remember that hiding secrets is hard." – Gary McGraw

Simulations are large pieces of expensive software that are algorithmic intensive and contain many numerical assumptions. In a coalition training and/or operational planning environment, sharing simulations requires an understanding of how much information is being shared.

The biggest customer of simulations is the United States Department of Defense [Balci 2001]. The United States shares these simulations with allies, but often does not want to share all of the training, tactics, procedures, battle management or other technical details with coalition partners.

The simple answer to this dilemma is to not release the source code of the simulations. The view being that only a binary executable of the simulation should be released; this is one step in the protection of simulations, but it ignores the fact that binary executables are vulnerable. A malicious user with enough time and resources can learn the internal workings of an executable without having the source code available.

This section 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.
9.4.1 HOW A MALICIOUS USER GAINS 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. Many techniques are available for the malevolent user to use to gain information. In this section we will discuss three, and also talk about one of the biggest security holes in software, that could also be viewed as vulnerability.

9.4.1.1 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 though, 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 gotten. The basic functionality of the program can also be determined.

9.4.1.2 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.
9.4.1.3 Monitoring Behavior

Monitoring the behavior of a simulation requires more patience than decompiling or disassembling the executable, but it has been known the 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 is 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. Because the simulation has multiple input values and output values, it would be difficult for a would-be reverse engineer to determine how they correspond. 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.

9.4.2 Buffer Overflows

Buffer overflows are a widespread problem in software development. The problem is simple; a programming language 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 user to allow an executable to be run with more permission 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.

9.4.3 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
Security Issues Resulting from Interoperability

don’t decompile the Windows OS, it is too massive to fully understand. A focused intelligence effort may need to compromise only some specific sensitive information.

9.4.3.1 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, such as turning debugging options off, will make it harder to decompile the executable. Also not including information that a user should be able to input, such as data, is a good idea.

9.4.3.2 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 the functionality is 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.

9.4.3.3 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 [Colberg 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 is not enough by itself. Layout transformation is the simplest form of obfuscation.

9.4.3.4 Data Transformation

Data transformations goal 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].

9.4.3.5 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 choose between a highly efficient program or a highly obfuscated program. The biggest issue with control transformation is making them computationally 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].

9.4.3.6 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.

JOBE and DashO are programs available to obfuscate Java class files. DashO proudly proclaims their use of a renaming system. Beyond that it seems to do no other obfuscation. This seems very much like snake oil, and developers should be wary of using obfuscation programs. KlassMaster is a Java obfuscator that does name and flow obfuscation. CShred is a program that obfuscates C files.

9.4.3.7 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 then 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 after
completion to encrypt the sensitive areas with a new key. This would make the
possibility for a brute force decryption attack even more unreasonable.

9.4.4 Client Server Model

A possible way to prevent reverse engineering of source code is to not allow
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 attainable 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 insure communication is secure and unmonitored now becomes the main
problem. One headache is exchanged for another.

9.4.5 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].
Joint Command & Control Interoperability: Cutting the Gordian Knot

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.

9.4.6 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.

9.4.7 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.

9.4.8 Keeping 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.
9.4.9 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.

9.4.10 Summary

“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. A practical example of this is demonstrated in the next section where a U.S. military simulation was red teamed and compromises reported to the sponsoring agency prior to foreign release.

9.5 RED TEAMING A SIMULATION BEFORE COALITION USE

The goal of this study was to determine the amount of sensitive information that could be derived from a military 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 the software, the analysis was conducted in four phases.

The first phase of the analysis consisted of attempting to find vulnerabilities in the simulation package 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 software, and analysis of the generated code. Particular attention was paid to any string constants contained in the disassembled code, along with the difficulty in the interpretation of the generated code.

The third phase in the analysis 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.

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

9.5.1 Phase 1: Simulation 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 designers of the simulation 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 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. Upon generating the segmentation fault, the currently executing code was examined via a debugger, more specifically the Microsoft Visual C++ 6.0 debugger as shown in Figure 9-9. 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 in the simulation 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.
9.5.2 Phase 2: Disassembly & Decompilation

The second phase of the study attempted to recreate the source code of the simulation software 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, this was considered particularly important. Clearly the extraction of the simulation source code from nothing more than the compiled executables must have an impact on the security classification of the model.

Preliminary efforts focused primarily on the decompilation of the source code. 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. 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 to be used on programs that were 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. 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. 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.

Furthermore, the Solaris version of REC generated only 19-24 megabytes of output code before crashing with a bus error. The sizes of these executables, along with the sizes of the resulting source code generated before REC crashed are as follows:

<table>
<thead>
<tr>
<th>Executable</th>
<th>Executable size</th>
<th>Source code at crash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable File 1</td>
<td>25792k</td>
<td>17,736k</td>
</tr>
<tr>
<td>Executable File 2</td>
<td>39440k</td>
<td>19,864k</td>
</tr>
<tr>
<td>Executable File GUI</td>
<td>37504k</td>
<td>23,000k</td>
</tr>
</tbody>
</table>

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, and is in an early stage of support for both the Windows and Solaris environments.

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. The package known as PE Explorer was tried first. PE Explorer provided valuable information concerning the 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 platform.

The Hackman application 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.

![Figure 9-10 PE Explorer Disassembly Package](image)

The disassembly of Hackman resulted in legal assembly code that was later re-assembled into an exact copy of the simulation executable. The simulation 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 “compunixsolaris” usenet newsgroup.
Three of the distributed executables were targeted for disassembly as shown in Table 9-3. The sizes of these executables, along with the sizes of the resulting assembly code are as follows:

<table>
<thead>
<tr>
<th>Executable</th>
<th>Executable size</th>
<th>Assembly size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable File 1</td>
<td>25792k</td>
<td>131624k</td>
</tr>
<tr>
<td>Executable File 2</td>
<td>39440k</td>
<td>141016k</td>
</tr>
<tr>
<td>Executable File 1GUI</td>
<td>37504k</td>
<td>167528k</td>
</tr>
</tbody>
</table>

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.

It should be noted that the failure to compromise the code in this examination does not mean that such an endeavor is not possible. An organization with sufficient funds would almost certainly be able to compromise the simulation code, 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, it would almost certainly be possible for an organization to implement their own custom decompiler specifically tasked with compromising a single executable. The only thing that would thwart such an effort would be the continued secrecy of the compilation platform used by the original developers.

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. In the case of the simulation, it was believed that the compiler used to create the Windows version was Microsoft Visual C++ 6.0, based on the information provided by PE Explorer. This belief was later confirmed by the simulation developers.

9.5.3 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 of the simulation 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 of the simulation.

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 executable. Just under 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 of the simulation 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. For example, by running the GUI program through Microsoft’s Visual C++ 6.0 Debugger, the names of several different functions could be found. In
addition to the function names, the number and type of arguments required by the function were also found. This could greatly assist anyone seeking to compromise the simulation code, even if they did not have access to anything other than the compiled executables. Table 9-4 contains information for all function names that were successfully extracted from the Windows version using nothing other than the MS Visual C++ 6.0 Debugger. It took less than 15 minutes to extract the preceding information. The amount of information that could be derived is probably much greater.

Table 9-4 Extracted Function Names

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>WinMain</td>
<td>HINSTANCE__, HINSTANCE__, char__, int</td>
</tr>
<tr>
<td>GetToken</td>
<td>None</td>
</tr>
<tr>
<td>SetScrollIndex</td>
<td>None</td>
</tr>
<tr>
<td>ReadOldRuleSets</td>
<td>_iobuf <em>, char</em>, char*, ScenarioHead*, long, SlewData*</td>
</tr>
<tr>
<td>MainWndProc</td>
<td>HWND__, unsigned int, unsigned int, long</td>
</tr>
</tbody>
</table>

As previously stated, the Solaris version of the simulation was not stripped of debugging information either. This means that attaching a debugger to the non-stripped executables will yield the ability to track variables as they are named in the original source code. This ability is drastically reduced by stripping this information from the executables.

In order to evaluate the differences between the stripped and un-stripped executables, the Solaris version of the simulation 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.

9.5.4 Phase 4: Source Code Analysis

As a final step in the analysis of the simulation, the source code was examined. The source code consisted primarily of the model itself, so that users could understand how different functions in the simulation was implemented. Two areas were of particular interest. The first area focused on finding security vulnerabilities that might allow a third party to take control of the executable. The second area focused primarily on the discovery of sensitive information contained in the source code.

After reviewing much of the code concerning input and output, it was determined that the application is reasonably secure. No unbounded buffer reads were found in the code available that could easily lead to a third party taking control of the application. The single vulnerability that was found in phase 1 was present in the GUI portion of the code, which was not available for review.

The second area of analysis for this source code review concerned the presence of sensitive information contained in the source code. Given the amount of source code to review, and the time allowed to review it, it was necessary to somewhat automate the process. The search was automated by use of the search utility present in Windows 2000 and a perl script written specifically for this project. The Windows 2000 search was utilized to locate keywords within the source code, such as the names of various platforms. The perl script was used to locate assignment and conditional statements present in the source code.

Many potentially sensitive pieces of information were derived from the keyword search. Unfortunately, without access to the classified data, and a thorough understanding of how each platform functions, it is not possible to tell the exact sensitivity of the information contained in the source code. Most of the data that was found dealt with comments that were present in the code, outlining improvements that were made. For example, a particular comment might state that, “Improvement X was made to support function Y of platform Z”. There were few instances of numerical data being present in the source code that might reveal performance specifications. Instead, it is more likely that a review of the
source code might lead to a greater understanding of how a particular platform works.

There were not many instances of actual data being hard coded in the source code. For example, almost all constants that were used were declared in files that were not included with the source code that was provided. Any constants that were hard coded appeared to be benign in nature, such as the default wind velocity for a simulation. That statement cannot be confirmed without direct access to the data, however.

9.5.5 Simulation Vulnerability Analysis Summary

After completion of a vulnerability analysis of the executables and the accompanying source code, several conclusions were reached. Therefore, it is entirely possible that some of the conclusions may be proven incorrect, based on information not available to the researchers.

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. The function and variable names alone are not very sensitive, though that information may aid an outside organization in their attempt to gain access to the code.

Third, it appears that neither version of the simulation had the debugging information stripped from the executables at compile time. This provides anyone great insight into both the names and purposes of the functions in the simulation. This information was quite easy to retrieve – the only necessary tool was a debugger.

9.6 CONCLUSION

Those pundits who glibly extol the virtues of commercial-of-the-shelf (COTS) hardware and software to the Department of Defense should seriously reconsider their arguments. The same COTS hardware and software enables potential
adversaries to rehearse attacks against DOD systems in the same manner that we used to assess our own vulnerability.

Modeling and simulation is clearly becoming a mainstream tool in many domains. It is simply too costly to flood operational networks and measure the results. A validated network simulation is clearly a more practical means of evaluation. Effectively modeling buffer overflow attacks does require a high fidelity model but it is clearly the best means for studying buffer overflows across network connections where the analyst only has access to one side of the network, such as on a Coalition WAN.

Not only are simulations useful for studying vulnerabilities, they are becoming a source of vulnerability themselves. Simulations are increasingly being used for coalition training and operational planning. Simulations suffer from the same vulnerabilities as any other sensitive software.

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. We believe a client server model with strong encryption authentication will be used to transfer sensitive information and protect US-only information.

9.7 REFERENCES


Joint Command & Control Interoperability: Cutting the Gordian Knot

