View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Calhoun, Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Theses and Dissertations Thesis Collection

2002-09

Delaying-type responses for use by software decoys

Julian, Donald P.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/5043

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DELAYING-TYPE RESPONSES FOR
USE BY SOFTWARE DECOYS

by

Donald P. Julian

September 2002

Thesis Co-Advisors: Neil C. Rowe
J. Bret Michael

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Delaying-Type Responses For Use By Software Decoys			**5. FUNDING NUMBERS**
6. AUTHOR(S) Donald P. Julian			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

Modern intrusion detection systems have become highly reliable in identifying a malicious user on a computer system. Their limitations, though, are increasing the need for an intelligent response to an intrusion. In contrast, intelligent software decoys provide autonomous software-based responses to identified intrusions. In this thesis, we explore conducting military deception, focusing on the use of software-driven simulations to respond to the actions of intruders. In particular, this thesis focuses on a model of a simple deceptive response that is intended to protect a search-type program from a buffer-overflow attack. During our study, we found that after identifying an attack attempt, simulating system saturation with processing delays worked well to deceive a prospective attacker. We also experimented with providing confusing reactions to an identified attack attempt, such as simulated network login screens and fake root-shells. The results were successful, simple reactions to intrusions that mimicked intended system interaction, and they proved to be adequate at implementing the deception principles we studied.

14. SUBJECT TERMS Intelligent Software Decoys, Intrusion Detection, Computer Deception, Decoy Response, Military Deception, Simple Deceptive Response			**15. NUMBER OF PAGES** 77
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 2-89)**
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DELAYING-TYPE RESPONSES FOR USE BY SOFTWARE DECOYS

Donald P. Julian
Major, United States Marine Corps
Engineering Physics (B.S.), Eastern Michigan University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2002

Author: Donald P. Julian

Approved by: Neil C. Rowe
Thesis Co-Advisor

J. Bret Michael
Thesis Co-Advisor

Lieutenant Commander Chris Eagle, United States Navy
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Modern intrusion detection systems have become highly reliable in identifying a malicious user on a computer system. Their limitations, though, are increasing the need for an intelligent response to an intrusion. In contrast, intelligent software decoys provide autonomous software-based responses to identified intrusions. In this thesis, we explore conducting military deception, focusing on the use of software-driven simulations to respond to the actions of intruders. In particular, this thesis focuses on a model of a simple deceptive response that is intended to protect a search-type program from a buffer-overflow attack. During our study, we found that after identifying an attack attempt, simulating system saturation with processing delays worked well to deceive a prospective attacker. We also experimented with providing confusing reactions to an identified attack attempt, such as simulated network login screens and fake root-shells. The results were successful, simple reactions to intrusions that mimicked intended system interaction, and they proved to be adequate at implementing the deception principles we studied.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION AND SCOPE ... 1
A. INTRODUCTION ... 1
B. BACKGROUND ... 1
C. SCOPE ... 3
II. DEVELOPMENT OF INTELLIGENT SOFTWARE DECOYS ... 5
A. BACKGROUND ... 5
1. Anomaly Identification ... 5
2. Misuse Detection ... 6
B. DECEPTION PRINCIPLES ... 6
1. Believability ... 7
2. Timely Feedback ... 7
3. Integration ... 7
4. Denial of True Activities ... 8
5. Realistic Response ... 8
6. Imagination ... 9
7. Hiding ... 9
8. Showing ... 10
C. SOFTWARE-BASED DECEPTION RESPONSE ... 11
1. Correctly Identify Attacks ... 11
2. Isolate Attacks ... 11
3. Believable Simulation ... 12
a. Predictable Reactions ... 12
b. Real-Time Interaction ... 12
III. ATTACKS AND RESPONSES ... 13
A. ATTACK PROFILE ... 13
B. POTENTIAL ATTACKER PROFILE ... 14
1. Categories of Attackers ... 15
2. Attention Span ... 16

3. Attacker Perception of Time……………………………………17
C. DECOY LEVELS OF RESPONSE……………………………………17
1. Simple-Level Response……………………………………18
2. Intermediate-Level Response……………………………………18
3. Complex Level-Response……………………………………18
D. STRATEGY OF A SIMPLE RESPONSE……………………………………19
1. What is a Simple Deceptive Response?……………………………………19
2. What is a Simulation?……………………………………19
3. How Should a Simple Response Be Developed?……………………………………20

IV. IMPLEMENTATION OF A SIMPLE DECEPTIVE RESPONSE……………………………………21
A. DEVELOPMENT……………………………………21
1. Normal Mode……………………………………21
2. Deception Mode……………………………………22
3. Timing Delays……………………………………23
a. Development Strategy……………………………………23
b. Precondition Checks……………………………………25
(1) Does the keyword string begin with "file//"?……………………………………25
(2) Does the keyword string begin with C-code?……………………………………25
(3) Does the string begin with a "//"?……………………………………26
(4) Is there only one long keyword entered?……………………………………26
(5) Are there more than ten keywords?……………………………………26
c. The Sleep Method……………………………………27
4. Login Screen……………………………………28
5. Root Shell Simulation……………………………………29
V. DECEPTION EXPERIMENT……………………………………31
A. INTRODUCTION……………………………………31
1. Subjects……………………………………31
2. Program……………………………………31
3. Method……………………………………32

B. RESULTS……………………………………………………..…………33
C. DISCUSSION OF RESULTS...……………………………………………35
D. CONCLUSIONS OF EXPERIMENT…………………………..………..36
VI. CONCLUSION AND RECOMMENDATIONS…………………..…………39
A. CONCLUSION………………………………………………..…………39
B. ACHIEVEMENTS OF RESEARCH………………………..………...39
C. LIMITATIONS OF RESEARCH………………………………………..40
D. RECOMMENDATIONS FOR FUTURE RESEARCH…………………40
APPENDIX A KEYWORD SEARCH PROGRAM WITH DECEPTION……...43
LIST OF REFERENCES………..……………...……………………………………..55
DISTRIBUTION LIST…………………………………………………………….….59

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1.1 The Greek Triad Model……………………………………………..…….….2
Figure 4.1 Normal Mode Diagram………………………………………………..22
Figure 4.2 Deception Mode Diagram……………………………………………..23
Figure 4.3 Keyword Search Table………………………………………………...24
Figure 4.4 Time To Search With Delay…………………...………………………...28
Figure 4.5 Login Screen……………………………….……………………29
Figure 4.6 Root Shell Simulation…………………………………………...….30
Figure 5.1 Sample Input Screen…………………………………………………..32
Figure 5.2 Message Box……………………………………………………….….32

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 5.1 Subjects…………………………………………………………………..31

Table 5.2 First Two Question Answers……………………………………………34

Table 5.3 Delay Perception…………………………………………………………34

Table 5.4 Reaction to Queries……………………………………………………....34

Table 5.5 Comments and Overall Rating…………………………………………...35

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENT

This thesis contains the contributions of many extremely intelligent people who brought immeasurable intellect to the weekly meetings of the Software Decoy Group. Dr. Neil Rowe, Dr. Bret Michael and Professor Richard Riehle, thank you for helping me focus this thesis on what it really is: the idea of three very insightful and inspiring computer scientists. Especially Dr. Rowe, thank you for your incredible patience and your passion to make computer science better.

This work is dedicated to my fiancée, Julie, whose love gives me the encouragement to want to be better every day. Also, to my children, Chelsea and Christopher, who are my inspiration and make me happy everyday just to say I am their dad. I will always work hard to make these three special people proud of me.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND SCOPE

A. INTRODUCTION

In the modern world of integrated computer communications, intrusion detection has become vital to the overall security of the individual networks. Many intrusion-detection systems have been developed and implemented to detect anomalous behavior. Firewalls, system logs, and virus detection have all been effective in identifying attack protocols and preventing potential intrusions. The usual response to these intrusions is a notification to the system administrator or filtering out of the network the source of the anomalous behavior. But what if information can be gained by allowing the intrusion in a controlled manner? Controlling the intrusion can be accomplished by crafting a response to a detected intrusion.

Two main levels of response are currently being explored. First, a low-level automated response using system-call delays is examined by Somayaji and Forrest in [6] as a way to allow an attacked computer to preserve its own integrity. The concept ignores the source and method of the intrusion and instead focuses on helping the affected system maintain a stable state. In contrast, Michael and Riehle have proposed an intelligent software decoy to deceive the attacker while maintaining system stability [1]. They define an intelligent software decoy as an object with a contract for which a violation of one or more preconditions by an agent causes the object to try to both deceive the agent into concluding that its violation of the contract has been successful and assess the nature of the violation, while enforcing all postconditions and class invariants [1]. In other words, they propose taking intrusion detection one step further by implementing intelligent responses and deception.

B. BACKGROUND

Computer security can be thought of as a silent alarm system that encompasses a computer network. Mechanisms should always be in place to protect the network, and to respond to a possible attack efficiently. Computer security is a *process*. As discussed by Wadlow in [2], the process can be applied again and again to improve the security of the

system. Wadlow compares the process of security to the ancient Greek triad-engineering model of Analyze, Synthesize, and Evaluate, as pictured below.

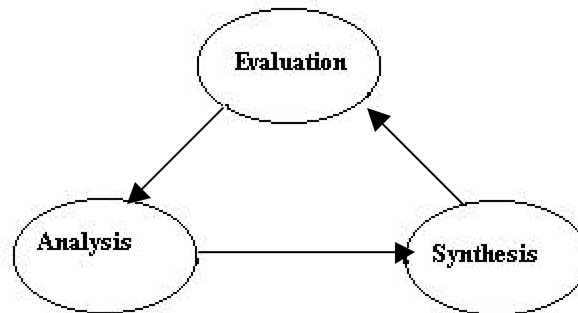

Figure 1.1 The Greek Triad Model

The success of the process, though, depends on the ability to synthesize and implement the ideas and concepts learned through the analysis of the network. This forces the need to gain as much knowledge as possible about possible computer-network attacks in particular.

Current intrusion-detection systems gain information about attacks and attack profiles, but they fall short at being dynamically adaptable and truly intelligent about their response to the attack. Most systems are capable of terminating connections, killing processes, and blocking certain messages from the network. As with most security devices, though, in practice these mechanisms cannot be widely deployed because of the threat, or risk, of false and inappropriate responses. Somayaji and Forrest suggest that the intrusion detection systems become too much of a burden to effectively analyze and respond to anomalies because of the overhead associated with human analysis [6]. It follows that an intelligent and autonomous solution is needed not only to protect the network, but also to gain information about current attacking techniques.

C. SCOPE

Our research focuses on exploring the level of response that is appropriate for an intelligent software decoy. We study three levels of response: simple, intermediate, and complex. Each level has unique development issues currently being explored, but they all relate back to the process of engineering the software decoy. The simple level of response seeks to exaggerate affects on the system caused by malicious use. By contrast, intermediate-level responses seek to respond to an attack by mimicking a previous attack. Finally, a complex-level response could simulate a successful attack on an entire computer system or network.

This thesis is limited to three primary tasks: discussing the concept of a software decoy, defining the levels of complexity of the responses, and examining the practicality of a decoy through the development of a model of a simple response. After we define each level of response, we develop a simple response model based on a Java Servlet program defending against a buffer overflow attack. By developing this proof-of-concept model, we hope to gain valuable insight into the development of more complex and functional software decoys.

THIS PAGE INTENTIONALLY LEFT BLANK

II. DEVELOPMENT OF INTELLIGENT SOFTWARE DECOYS

A. BACKGROUND

With the always-increasing dependence on computer systems and computer networks in today's society, application possibilities are limitless. Unfortunately, the risk of malicious use and information compromise is also increasing. Modern computer systems must be designed to prevent unauthorized access to the resources and data they contain.

Intrusion can be defined as a deliberate attempt to gain access to unauthorized information, to change or manipulate information, or to make a system unstable and unusable. All are attempts to degrade the computer system's to provide for desired levels of confidentiality, integrity, availability, and other security properties.

While intrusion-detection systems have been studied exhaustively, they are still reactive rather than pro-active to intrusions. For instance, one of the most popular and widely used ways that intrusions are detected is through audit-trail use. This involves studying each event that happens on a system as recorded through the continuous logging of events. The amount of data collected per day could be quite large, especially if one instruments a system to monitor many events that occur on a frequent basis. Software, however, has been developed to allow the quick studying of events and to look for anomalous behavior. The two main ways to discover an intrusion are *anomaly identification* and *misuse detection*.

1. Anomaly Identification

Behavior in a computer system can be best described by defined functionality of a user within a system. Each user has pre-defined rules that govern their actions while a member of that computer domain. These rules are based on statistical profiles that are developed for individual computer users [13]. The current behavior of the user is then compared with recorded behavior to determine whether it is normal. Identification of anomalies can potentially recognize unforeseen attacks, although it is not foolproof. Anomaly identification can be limited by vague rules which make it difficult to distinguish between normal and abnormal behavior.

For example, a trusted user may have the privilege to modify a particular file, while an untrusted user who is accessing the file over a network connection may only have permission to read the file. In this case, an anomaly would occur if the untrusted user tried to modify or change the permissions of the file to give him access so he can then modify the contents.

2. Misuse Detection

Intrusions can be described in terms of the indications and signs they leave behind on the system. Misuse is based on expert knowledge of patterns associated with unauthorized use or activity [13]. These patterns, sometimes called signatures, are compiled and then implemented into detection programs that find a match between a signature and current activity. Pattern analysis can also be used to study logs and system-audit information. Once a pattern is identified in the logs, it can be described abstractly to allow its recognition in the future.

The principles employed with intrusion detection systems are widely used today to protect information systems. They supplement tools such as firewalls, encryption, and authentication.

B. DECEPTION PRINCIPLES

Sun Tzu makes a simple yet profound statement in *The Art of War* that "All warfare is based on deception"[8]. An argument certainly can be made that computer security, and indeed information assurance, is a part of future warfare. Can deception help protect systems? Can it encourage an enemy's arrogance by feigning a poorly protected computer network?

Deception fundamentals are provided from other basic military operational techniques. The following are the six basic rules of deception that could be applied to computer deception [7]. We also include two other basic concepts of deception and cheating discussed in [9]: hiding and showing.

1. Believability

The effectiveness of deception is first based on the ability to have the enemy believe what he is experiencing is real. That is, to make an action appear to coincide with the enemy's preconceived notion of the expected results. For computer security, the enemy is the malicious user who is attempting to access a system without the proper permissions. To make the deceptions successful, the malicious user must believe he is accessing the system. The correct simulated reaction to attacker input, generated directories that appear to have accepted malicious code, and appropriate delays to simulate compilation may make the simulation of a computer attack believable.

2. Timely Feedback

Timely feedback is an essential element of all major deceptions. It is important to observe the reaction to any deceptive activity because if the attacker does not accept the deception, the system running the simulation may be vulnerable. In an autonomous intrusion-detection system, real-time observation is delegated to the logic of the detection program. This allows observation of the reaction to the deceptive activities. To make this feedback effective for an intrusion-detection system, a good study of what beliefs and biases are built into the attacker culture is necessary to improve the likelihood that one has created the correct deception to anticipate the reaction with little error. This includes a study of the timeliness of actual deceptive activities. If a computer simulation reacts too quickly or too slowly the activity may not fool the attacker. Therefore, the simulation of system effects of an attack must occur in accordance with the time the attacker expects.

3. Integration

The plan for deception should be integrated with the goals of the operation; in particular the normal processing of a computer system should include the well-formed plan for deception. The operation and deception plans must be mutually supportive and should complement each other. This will not allow for ad-libbed deceptive operations that have been shown to be counterproductive [7]. When the deception is integrated properly into the operational plan, the overall effect will provide believable indicators of the false operations and will deny believable indicators of the real operation. Integrating

the deception within the software of a system, as opposed to using a separate Honeypot-type system [18], may allow for a more effective way to use computers to protect themselves against attacks vice dedicating separate networks to catch and analyze intrusions.

4. Denial of True Activities

The deception must be stealthy and should protect the true activities. In computer system terms, a carefully constructed software-based deception could effectively deceive an attacker, but only if the attacker does not receive contradictory information from the real activity of the system. One way to suppress the actual operating system functionality would be to isolate the attacker so he has no ability to interact with the real operating system. Isolating the attacker in the software is accomplished by coordinating system delays in the place of actual system interaction. Another way to protect the true activities of a computer system is to portray the time a process takes to complete while not actually performing the task. For instance, if a search program takes a certain amount of time to accomplish, then a simulation of that search should delay the user for that same time. Delays can be used effectively to simulate system interaction thereby denying the actual activity of the computer.

5. Realistic Response

Deception should be a function of the level of response desired to ensure the proper degree of realism. There should be a level of deception that is proportional to the time that is needed to analyze a situation and take appropriate action. For example, if an attacker is simply attempting to launch a denial of service program on a system, it may be sufficient only to delay for the amount of time it takes to compile the program. It may not be necessary to carry the deception past the simple delay. However, if an attacker tries to place a root kit program on a target, it would be necessary to simulate a root-shell screen with input and output capabilities. The root-shell should also be supported by a simulated complex file-system to make the response as realistic and believable as possible.

6. Imagination

History has shown that the most successful deceptions have been imaginative and creative. When a deception becomes too predictable or stereotyped, the effectiveness may be compromised. Standard deceptive responses should not be developed and should not imply the development of a capability that has not been realized. Deception is best accomplished by utilizing inventive and bright ideas. Imaginative computer deceptions can include creative error messages, generated file systems that are viewable to the attacker, and the appearance of new interactions. The deceptions should, however, follow along with what is expected to ensure the believability of the simulation. On the other hand, if an attacker finds he is presented with an unexpected way to gain access to a computer system, he may be more inclined to attempt a breach. Some other creative ways to keep an attacker's attention could be to present file systems that are seemingly retrievable, simulate the allowance of an attacker to place files on a system then modify them, and also to create an environment that simulates root-access that is receptive to input and output. The imagination of the simulation programmer is unlimited, and the development of software can support nearly any kind of simulation.

7. Hiding

According to [9], the psychology of deception uses two basic concepts of the devices found in nature. The first concept, hiding, fulfills the basic purpose of deception to screen the thing you mean to protect. Altering and covering the particular details of the thing, in our case a computer system, can be used to gain some advantage over a potential adversary. For example, we can hide an important file on a computer system by saving the file under a different name among some other bogus files with non-descriptive names. Hiding computer system information can also be thought of much like hiding a submarine in a vast ocean. Software decoys, in general, will avoid hiding themselves on a system, but they can be used to hide the actual operating-system functions. For example, decoys can hide the fact that they are delaying a user, to simulate system interaction, by mimicking what really happens when a user interacts with an operating system.

8. Showing

Showing is a deception where an altered truth is presented. There are three ways to show the false: by mimicking, inventing, or decoying. A replica of reality is created by selecting one or more characteristics of the real to achieve an advantage when mimicking [9]. There are many classic military examples of mimicking. For instance, the Germans during World War I portrayed to the Belgians that their force was more than 150,000 troops when it was indeed less than 20,000. The German mimic was a success because there was an extended period of time when the illusion was not revealed, allowing them to gain an advantage. Mimicking with software is accomplished with a Honey Pot-type system that pretends to be an unprotected computer system or network and seems to be easier prey for intruders than true production systems with minor system modifications [20].

Inventing, on the other hand, displays through the fashioning of an alternative reality. For instance, a false file created on a system is still a file, and it does not mimic a real file. The goal when inventing is to create something that has the effect of being real to ultimately conceal the real. Invention in computer systems can be accomplished with scripted games where the input from an attacker is anticipated, and the intended result is simulated to look real. In this way, an alternate cyber-world is created that seems interactive, but is actually only responding in a predetermined way.

Finally, decoying can be simply defined as deploying some unified type of defense with a variety of potential directions and possibilities. When a decoy is properly implemented, it can hide the real intentions with false options. In military operations, decoying attempts to mislead the time or direction of a particular attack. Computer systems can accomplish decoying by simulating that an attack is actually functioning as intended, but without actual operating system interaction. Computer system decoying will most effectively be accomplished by planting simulation routines into the software that will simulate attacks when triggered. This type of decoying is in contrast to the Honey Pot vision discussed in [20] in that it is integrated into the system through the use of intelligent software and preconditions.

C. SOFTWARE-BASED DECEPTION RESPONSE

A "software decoy," as proposed by Michael and Riehle in [1], seeks to develop the capabilities of intelligent software modules to react to malicious use. For a software decoy, we propose that there are three main goals of a deception: correct identification of malicious behavior, isolation of the attacker, and believable simulation of the attack effects.

1. Correctly Identify Attacks

As previously discussed, accurate and timely detection of anomalous behavior is the cornerstone of the success of the deception. Current intrusion-detection systems can identify malicious use. Can they identify all attacks? Not yet, but many rules defining "acceptable" behavior exist and should be used. These rules have well-formed thresholds that are set high enough to allow for a high degree of confidence in their capabilities to correctly identify attack protocols. With any detection system, for example with biometrics, obtaining false positives and negatives are a possibility. A set of preconditions should be developed for software decoys that will allow a high degree of confidence in the detection logic.

2. Isolate Attacks

Simple decoys can extend a particular program; if certain behavior is observed, the program can behave in a way other than what is intended. In fact, the program is behaving how the programmer envisioned, but the system accessibility of the user is being carefully controlled by the program logic. The malicious user can be thought of as operating in a chamber that looks and acts like the operating system he expects, but is actually an isolated part of the program that cannot interact with the actual system. How an attacker can be isolated is up to the programmer. For instance, the programmer could offer a root-shell that looks and acts interactive, but is actually a data-collection box that has no ability to send commands to the processor other than "Quit." While the isolation logic must be completely foolproof, so there is no threat of unintended system interaction, it is, nonetheless, essential to a decoy.

3. Believable Simulation

The success or failure of the decoy hinges on the ability to deceive the attacker. There are two main concepts to make the simulation believable to keep the attacker occupied and away from the actual system: make the reactions predictable and make the interaction real-time.

a. Predictable Reactions

Thorough research into the expected results of a given attack must be done. The simulated system response should be in line with what the intruder is intending. For example, when a root kit is successfully installed on a computer with a Trojan horse, the attacker will expect access to the system as a root user with root privileges. The deception should account for this by presenting a simulated root shell that the attacker can use; otherwise the reaction will not be predictable and could fail. It may also, however, be interesting to an attacker to find some confusing and unexpected results to an attack attempt. The reaction may inspire new attempts and create new protocols from which new intrusion detection signatures can be developed.

b. Real-Time Interaction

Beyond protecting the operating system, the main goal of a decoy is to keep the attacker occupied and away from the main system. To keep the attacker occupied, the interaction should be real-time. If each step of the simulation needs to be compiled or logically determined, the delay may cause the attacker to become uninterested or, worse yet, uncover the simulation. For instance, if the attacker tries to cause a denial of service by flooding the network, every command that the attacker (and other users, too) tries to execute must be delayed accordingly. This real-time interaction is necessary to allow the deception to continue.

III. ATTACKS AND RESPONSES

A. ATTACK PROFILE

For this study, we narrowed our scope of possible computer exploits to a well-known attack profile. We chose a buffer overflow exploit because of the large amount of documentation available on the protocol of the attack. A buffer overflow attack is a classic exploitation in which a malicious user sends a large amount of data to a server to crash the system. The system component contains a buffer of fixed size in which to store this data. If the amount of data received is larger than the buffer, parts of the data will overflow onto the stack. If the component does not properly handle the resulting exception that is raised, a security breach is possible. By tricking a program into loading machine code into its memory, it is possible to overwrite the return pointer of the function [10]. If this data is code, the system will then execute any code that overflows onto the stack.

Buffer overflow attacks exploit the lack of bounds checking on the size of input being stored in a buffer array. By writing data past the end of an allocated array, the attacker can make arbitrary changes to program state stored adjacent to the array. In practice, the most common buffer overflow technique is to exploit the weakness by attacking the buffers located on the stack itself. Since the program input comes from a network connection, and since file processing often involves temporary changes in access rights, the class of vulnerability may allow any user anywhere on the network to become a root user on a local host. This makes buffer overflow attack identification critical to practical system security.

When the attacker is seeking to inject his attack code, he provides an input string that is actually executable code. The code is primarily binary machine code that is native to the operating system being attacked, and it can be simple. A common attack in Linux systems produces a basic shell which the attacker can then use to enter the system. The injected attack code is a short sequence of instructions that creates a shell, under the user-ID of ***root***. The effect is to give the attacker a shell with ***root*** privileges.

Normally, the targeted program would not execute malicious code that is entered as input, through a string. However, if that input is sufficiently long, the buffer can

overflow because the input to the program is placed on top of the process stack, where the computer is keeping track of the program's input and output [13]. When a program function is invoked, a return address to the next code to be executed is placed on top of the stack. The input buffer is placed on top of that address allowing for the opportunity to write into that space with the input data. Using this technique, it is possible to overwrite the return address with a malicious address. This allows the attacker to control the jumping of the function back to the attack code instead of the point where the function call was made.

The process of engineering a buffer overflow attack is not simple, but can be done by an attacker reverse-engineering the targeted program. This is necessary to determine the exact offset that the buffer needs to compute the return address in the stack frame. Sometimes, to lessen the complexity of engineering an attack, an attacker estimates the return address, then repeats the desired return address several times within the approxamate range of the current return address.

The description of the attack techniques makes it sound as if exploitation through construction of a buffer overflow were quite straightforward. The real work, however, is finding a poorly protected buffer in which to attack. Many systems have vulnerabilities that lend themselves to this and many other types of attack, in fact. In 1997, it was reported that the most common vulnerabilities on the Internet were buffer overflow and shell escapes [13]. A shell escape incorporates malicious code into the input data for a program following an escape character within a sequence of commands. In some cases, when a system encounters the escape character, it invokes a shell program to interpret and handle the code. Both of these vulnerabilities, whether used separately or together, result from the inadequate checking of input data.

B. POTENTIAL ATTACKER PROFILE

The types of deceptions discussed so far are not specifically aimed at one type of attacker. To better defend against a potential attack, it is important to understand the mindset and the intentions of a potential attacker. Moreover, it is also prudent to understand their perception of reality and their overall attention span to effectively

develop responses to their malicious attempts. Overall, an attacker can be characterized, as in 1999 by the Federal Bureau of Investigation [16], as a nerdy, teen whiz-kid who may be an antisocial underachiever, or maybe even a social guru. Their style is different because they think differently, and their intentions may vary from using hacking as a social and educational activity to serious acts of fraud, sabotage or espionage. This may be attributed to the fact that most are teenage males who are proficient in C-programming language, has a good knowledge of the TCP/IP protocol, and is usually intimately familiar with UNIX. All of the attributes may indeed just be a profile for a regular teenager, but it is the attacker intentions that set them apart. There are many types of attacker, and they are typically characterized by four terms: hacker, cracker, phreak, and cyberpunk [16].

1. Categories of Attackers

First, a *hacker* is typically a person who is very computer intelligent and who often enjoys examining operating system and application code to discover how it functions. This type of attacker uses his or her skill to penetrate software systems without permission and tamper with the data contained on them. The biggest threat from a hacker is their ability to implement espionage techniques for illicit purposes, such as economical gain or even cyber-terrorism.

A *cracker* is popularly characterized as a user who circumvents security measures of a system in order to gain unauthorized access. Their main goal is to break into a system without tampering with data or employing espionage techniques. These types of attackers are likely to seek use of the systems resources, possibly to launch a denial-of-service attack at a later time. A cracker is also capable of being a governmental hacker, where the targets are government computers and cyber-terrorism or cyber-warfare is the ultimate goal.

A *phreak*, with the unusual spelling, is a person who simply is trying to use a network illegally, that is, without paying for the service. These attackers have been observed as far back as the 1970s when phone phreaks would try to gain access to long-distance networks using their own hardware to match the tones on the phone line.

Phreaks may only be interested in gaining access to elevate their own social status or acceptance, similar to a graffiti artist [16].

Finally, the last category of attacker, a *cyberpunk*, or *script kiddie*, is a combination of all of the above types of attacker. Their mutation is growing from the proliferation of the Internet, and they may be the most common and dangerous type of attacker found today. The cyberpunk may have many possible goals in mind from gaining social acceptance to committing robbery or to, ultimately, launching a cyber-terrorist attack at a target, whether against the government or business.

2. Attention Span

Considering all of the attributes of an attacker, what their intentions are, and how smart they really are, how much time do they actually spend trying to attack? This may be too broad of a question for the scope of this thesis, but we are interested in studying how an attacker perceives time, and how much time he will invest to attempt his attack.

As a an economist once stated, "What Information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention" [17]. As far as hacking is concerned, this suggests that the time an attacker spends attempting access or exploiting a website is proportional to his desire to actually succeed. If he tries to gain access, but is thwarted by a firewall for example, he may give up and try to find another victim. Remember, most of the time the hacker is trying to gain not only access, but also peer recognition and popularity.

As far as a simple deception that manipulates time is concerned, then, how long is too long and how long is not long enough? Research gained from the Honeynet Project [18] shows that there are advantages to speed for an attacker. The extent of the advantage, though, is relative to the time a defender takes to detect and react to the attack. There is a strong advantage to the attacker if he can gain information about the defenses of a system. If he is able to determine the defenses, by finding the reaction times somehow, it is likely he be able to develop a strategy for overcoming the defenses. Attackers will do this by attacking and then observing the reaction to the attack. Therefore, the deception needs to be unpredictable enough in the sense that it cannot be

traced, but predictable enough, in terms of reaction times, to not seem like a simulation and seem believable.

3. Attacker Perception of Time

It is commonly recognized that people perceive changes or events in time, not precisely time [19]. That is to say, it seems logical to perceive one event after another event, not the time it takes to complete the event. The duration of the event is possibly more complex to understand. However, what is really being measured or perceived is the duration of the event in the memory of the perceiver. It is the memory of a previous hack, or a belief of how a hack will work, that drives the realization of the duration to a hacker that his attempt has been successful.

For the purposes of deception, an attacker perceives time on two levels. First, when he launches an attack, he estimates, based on a previous attempt, how long it will take to process a request on a given system. This knowledge may also be based on a hacker attempting small requests and observing how long each process takes. The second level of time perception is purely event-driven. That is, the attacker is only concerned about what happens next in the attack, similar to a script. For this second level, he may not be concerned with the amount of time it takes to process a request or a malicious request, but he is only concerned with the end result. For these reasons, it is important that the simple response model be developed with a firm grasp of processing time in order to implement the correct exaggerated delay into the simulation.

C. DECOY LEVELS OF RESPONSE

There are many different kinds of computer security, but the common thread among all security types is that they are proportional to the value of the system they are protecting. For example, computer systems that process air-traffic control need to be protected much more than systems that processes flower orders. Both systems need protection, to be sure, but the level of protection is vastly different. The engineering of the levels of protection, because security is a process, serves as a basis for the development of the response. What degree of simulation or deception is desired for the system and what the software decoy will protect are important questions to answer, and

the answer will drive the development of the deceiving software. Three possible levels of decoy response are possible, each with its own advantages and limitations: a simple-level, an intermediate-level, and a complex-level response [1].

1. Simple-Level Response

A simple response is a context-free response and can often be an exaggeration of system effects. Additional routines can be written into software that will react in a predictable way when a malicious user attempts to attack the program. This could be as simple as inserting exaggerated sleep time, where the user thread is rendered inactive for a certain period of time, into the processing of a request or it could be deceptive "games" that entice the attacker to stay occupied in the deception. My study included developing a decoy that delays users who are identified as malicious.

2. Intermediate-Level Response

An intermediate-level response is a step above the simple response in that it could run canned scripts mimicking what the attacker would expect to see when he launches a specific attack. This would require a database of known attack conditions that could be evaluated to determine an attack is in progress. The proper decoy or deception program could then be invoked. The same principles of isolation and exaggeration would then apply, but the degree of difficulty of the deception could be much more complex than the simple-level response. For example, if the attacker tried to run code that enabled a root shell to launch, the decoy could deploy a simulated shell and keep the attacker isolated from the system. We experimented with creating a root-shell to mimic a Unix operating system that could be deployed during an identified attack, and the results are discussed in the next chapter.

3. Complex-Level Response

The third level of placement is the complex-level. Decoys at this level could involve simulation of operating systems and networks with dynamic and distributed affects of an attack. The principles of the first two response levels still apply here, that the attack be recognized and isolated. A simulation could be run from a Honeynet-type [18] system that can broadcast simulated attack responses on a broad scale. Of course, it depends how important the network is to protect as to how elaborate the decoy should be.

It may be efficient to simulate an entire network, possibly on a CD-ROM. This level of decoying would provide many options for an attacker, and may expose some new information that could not be gathered when the simulation was limited to simulated scripts and time altering. Decoying at this level has been investigated in [23].

D. STRATEGY OF A SIMPLE RESPONSE

Sun Tzu, in *The Art of War,* also encourages his generals to feign incapacity when capable, and to seem inactive when active [8]. With software-based deception, these concepts suggest responses at all of the defined response levels. However, the incapacity must be thoroughly thought out logically before it is implemented. The idea is to anticipate the logic of the malicious user to correctly lead him down our deceptive path. This is attainable with thorough research into the anatomy of a simple attack. What a simple response is, how a simulation is defined, and how a simple response should be developed are all questions that need to be answered to effectively develop a simple response.

1. What is a Simple Deceptive Response?

A simple deceptive response is an attempt to simulate the effects of a malicious user on a system, often exaggerating the intended results. It should be implemented into the software code, and it should behave in a predictable manner. One simple deceptive response, as previously discussed, could be a processing delay that is implemented into each request an identified malicious user makes. The simple level of response is envisioned as a self-protective measure embedded into software. If its behavior is developed correctly, the response will be believable and will protect the system without using the system resources. In other words, when an attack is realized, the software logic handles the malicious request by running another portion of the program, while continuing to provide service to other users. A simple response is a reaction to a simple attack.

2. What is a Simulation?

Simulation may be the most powerful misdirection tool in computer system design, especially when it is combined with the development of the human interface.

Professional designers of computer systems create a world where interactions happen in a predictable and familiar way. Simulation of these interactions can be analogous to a magic act. For example, the mechanical devices and techniques that support a magic act are virtually transparent to an audience member. However, there are actually two simultaneous acts taking place when a magician is performing: the magician's reality, and the audience members' reality. The magician perceives his own sleight of hand and manipulative devices. The audience, though, has an entirely different view as long as the magician is doing a competent job [14]. This alternate reality is often where the normal and predictable laws are violated and defied. Human and computer interaction in a simulation is developed to make the user believe that one particular thing is happening, but in fact something else is taking place. A simulation is, in terms of computer interaction, the careful altering of a users reality.

3. How Should a Simple Response Be Developed?

A simple deceptive response should be developed with some limitations according to a stated policy. For example, Sun Tzu advises his generals to pretend they are inferior to encourage their enemy's arrogance [8]. This could have adverse consequences because it could charge the attacker with vigor to defeat the deception. Keeping this in mind, I studied simple deceptive responses using a variety of ideas. Some responses are minor, some intentionally misleading and confusing. The employment strategy, though, must be clear with the potential consequences of perceived inferiority thoroughly examined before implementation. Even with this simple response, a clear policy must be in place to determine the degree of simulation that is to be employed.

IV. IMPLEMENTATION OF A SIMPLE DECEPTIVE RESPONSE

A. DEVELOPMENT

Development of the simple deceptive response incorporated all of the research of attacker logic as well as recognition of the limitation of the extent of the deception. While the research is ongoing, our intent is to demonstrate the concept of a simple decoy response for software-based deception with a variety of logic-based ideas. As a starting point, I first state the logic of the test program by defining normal and deception mode. I then explore three possible methods of implementing a simple deceptive response to a malicious attempt to overflow the input buffer by examining timing delays, simulated logon screens, and simulating a root-shell.

1. Normal Mode

To develop a simple deceptive response, we began by modifying a program that was created for support of The MARIE Project [21]. The program is a Java Servlet that provides the user interface to the MARIE-4 system. The basic operation of the program is that it accepts a list of keywords from a user via a front-end web page. The web page is connected to a Jakarta Tomcat web server [22] on a Unix operating system. The keywords are parsed into words using a *StringTokenizer* Java method [15]. The program searches a database in a locally stored file to find rated caption candidates that match the keywords. The matched pages are then listed in decreasing order of rating. Normal mode is explicitly defined as a user accessing the web page, entering a string of keywords, and then receiving the requested pictures and links through the browser-based interface. On the top of the next page is Figure 4.1 that shows normal operation of the program in detail.

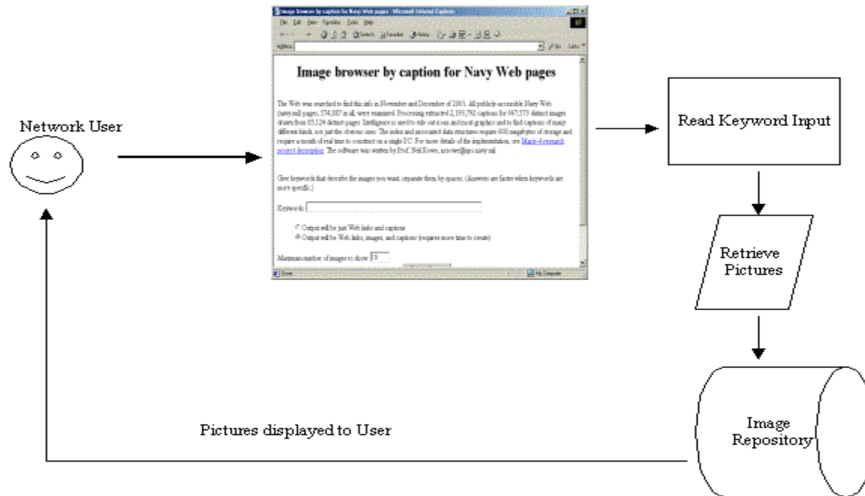

Figure 4.1 Normal Mode Diagram

2. Deception Mode

To operate in deception mode, there first needs to be a trigger inserted into the body of the code to allow access to the simulation once any number of abstract preconditions are met. Once the preconditions are met, the deception is triggered, and the program is now operating in deception mode. That is, the visual portion of the program looks real to the user, but it is actually responding differently in a simulated way. If the preconditions are not met, the program will continue to operate in normal mode as pictured in Figure 4.2.

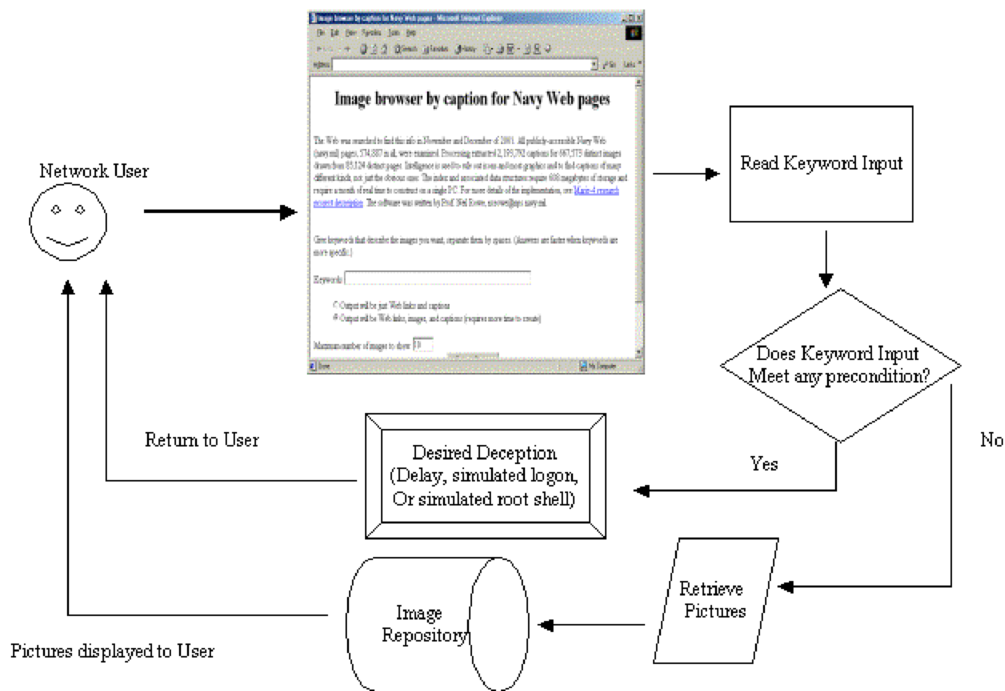

Figure 4.2 Deception Mode Diagram

Note that if deception mode is triggered, the operating system cannot be attacked because the simulation will run completely within the program. That is, the simulation is being run in a virtual chamber thereby protecting the operating system.

3. Timing Delays

The development of timing delays for deception was in three phases. First, we studied the strategy of the delays, then how to develop abstract precondition checks. Next, we designed and implemented of the notion of "Sleep."

a. Development Strategy

My experimentation with timing delays tested the idea of simulating an increased load on a computer system. We began with the study of a buffer-overflow attack on a web-based search engine. The goal was to exaggerate the delays caused by an attempted denial-of-service attack. For the deception to be successful, the thread

(process, which could represent a user) that is identified as malicious should be the only user or thread to get the exaggeration of the delay. The servicing of other users who do not meet the malicious criteria should continue uninterrupted. In modifying the original program, we tried to simulate the time that the image-library server would pretend to be busy for an identified malicious user. The logic is that the overflowed buffer caused by the malicious user is bogging down the response of the computer system. By delaying the request from the user, we can simulate a successful launching of an attack.

Using the previous discussion of the time perception of a typical attacker, we reasoned that the delay should be proportional to the average time a program would take to run on a targeted system. This time is variable due to the number of times the program has run and the amount of network traffic present. In practice, the first time the program was called it took significantly longer to retrieve the pictures and links compared to other times it was called from the web link. Therefore, the delay time needs to be proportional to the expected perceived time to process the malicious requests after the first one. Also, if the keywords entered were vague, such as the words "sea" or "ship," the search took considerably longer to complete and was aborted after five minutes. Figure 4.3 below shows the results of a search for some common words, while operating in normal mode.

KeyWord	**Times in Database**	**Show Results**	**Time to Search**
Plane	70792	918	3 sec
Ship	727291	Undetermined	150 sec
Aircraft carrier	2392 / 13003	449	3 sec
Destroyer	37994	541	5 sec
Submarine	11721	332	2 sec
Helicopter	1068	31	2sec
Surface	9157	269	1
Jet	1877	55	1
Sunset	1824	41	1
Sea	1164373	Undetermined	Undetermined
Blue Angels	2226 / 487	76	1
		Avg Time	**2.1 sec**

Figure 4.3 Keyword Search Table

The amount of search time in Figure 4.3 was used to help determine the time the program would need to search for keywords, so that the delay time could be added proportionally. The results of the delay are displayed in Figure 4.4.

b. Precondition Checks

First, to identify a malicious user, we added a number of precondition checks to the program to evaluate whether a user was attempting to launch a buffer-overflow attack. All preconditions were evaluated after the keyword string was broken down into words with the *StringTokenizer* method [15] and before the words were matched in the database. We assign a weight factor to each precondition to allow for different degrees of suspicion depending on the likelihood that the attack is really an attack vice a typographical error. We started by creating strings of known attack beginnings as a comparison for the keyword strings. Then, we asked the following questions:

(1) Does the keyword string begin with "file//"? If the string begins so, there might be an attack impending since the text contains escape characters. But the string could also have been a typographical error, and the user may not be attempting an attack. As a result, we assign the likelihood of attack a value of 1, which will still trigger the delay, but the program will still attempt to search for the pictures after the delay.

(2) Does the keyword string begin with C-code? This check is a little more complex, but the likelihood of attack is higher if it succeeds. First, we determine if the keyword string begins with an expression that resembles the beginning of a C-code program, such as "#define." Then, we determine the number of words, or tokens, that are present in the string. We reason that if there are many words, the likelihood of a malicious program is larger because a program contains many tokens of varying length. For this precondition, we choose twenty tokens as a starting point. If there are more than twenty words present in the string we assign a value of ten to the likelihood variable. If there are between ten and twenty words there is still a moderate possibility that the string is a program and the likelihood variable is assigned a value of two. Coupled with the first check of "#define," we reason that either length of string is

an indicator of a possible attack. There is one difference in the reaction. A moderate likelihood of attack will delay the user, then display the results of the search. A probability assignment of ten, however, will delay the user for a long period of time then close the connection by exiting the program

(3) Does the string begin with a "//"? Then we evaluate the length of the keyword string. The theory is that if there is an escape sequence followed by a long string of characters, the likelihood is high of a buffer-overflow attack attempt. A minimum keyword string length of 100 characters was chosen to be sure there is an attack underway if there are that many characters present. If the number of characters exceeds 100, we assign a delay probability of ten and send the user to an unrecoverable sleep sequence that will terminate the connection. The slash combination, however, may in fact be a typographical error not intended to launch an attack if the string is not 100 characters long, so we assign a delay probability of one, then retrieve the pictures from the repository, knowing that other logic in the program will prevent a search for non-English code-name words.

(4) Is there only one long keyword entered? If that word has a length greater than 50 characters, we reason that the probability of attack is moderately high, and assign a value of seven. The program delays for a long time, and the server will close the connection before any pictures are retrieved. If the count of words is again only one, but the string is less than 50, the user may have forgotten spaces between the words. For this reason, we allow a search without delaying knowing that the word may not be valid at which time the only result will be an error statement output to the user.

(5) Are there more than ten keywords? If more than ten tokens are entered, we reason that there is a low probability of attack. The delay factor in this case will be small (one), but compared to the length of delay for the program to search for the pictures it is not significant. The reasoning is that if an attack were attempted, there would probably be other triggers present that would be determined by the previous checks.

c. *The Sleep Method*

Once the triggers were declared, the actual delay function of the program was developed. The most straightforward way to delay is to put a suspicious user thread in a Java-defined ***sleep*** mode for a certain amount of time. The delay should also have a randomness about it that could be modified by the code calling the method. And lastly, the code should have two modes of operation, one that eventually returns the user to normal mode, and one that terminates the user thread without returning.

Putting a user thread to sleep is a simple, yet effective way to delay the user. The Java ***sleep*** method, when called on a running thread, forces the thread into a non-active, or sleeping, state. A sleeping thread cannot use a processor even if one is available [15]. The sleeping thread only becomes ready after the designated sleep time expires.

The code should be able to randomly generate a sleep time based on the likelihood the tagged thread is a malicious thread. One of the principles of deception is that it should not be predictable. Randomizing the delay function is a good way to make the delay unpredictable. In our program, the assigned likelihood of attack is given a value from one to ten, as discussed above. This value is passed into the ***Sleep*** method as an integer. If the integer is not the trigger value of 86, the ***Sleep*** method generates a random integer, between 1000 and 10000 representing the initial milliseconds the thread will sleep, using the ***random*** Java method, and multiplies that number by the likelihood value. The thread is then put to sleep for that many milliseconds. The range of values the user thread can be put to sleep is from 1 second to 100 seconds to parallel the time the system would be busy processing a malicious request. As displayed in Figure 4.3, the average time for processing keywords is 2.1 seconds. To properly mimic this time, we add time to simulate network congestion and possible processing queues. Therefore, the minimum time for delay should be approximately two seconds, while the maximum time of 100 seconds will simulate a large amount of processing time based on the successful launch of an attack.

If the number passed into ***Sleep*** is a trigger value of 86, the method enters what we call the "***deathSleep***" which puts the user thread to sleep for 1000 seconds and

then exits without actually performing the search. This is for when the likelihood variable has been given a maximum value of ten, and is intended to simulate a crash of the system. Currently the value of 86 is assigned only when the input keyword string begins with "#define" and continues with an amount of words greater than or equal to twenty.

Figure 4.4 displays the delaying aspect graphically. The graph plots the time the program takes to display the correct results while incorporating the desired likelihood of attack, designated in Appendix A as the "*weightOfAttack*." The results were obtained by entering the same eleven keywords to trigger the delay while varying the *weightOfAttack* factor. There were three series of data collected, each over a span of five search attempts. Series 1 represents the time to search with a *weightOfAttack* value of one (1), Series 2 uses a value of three (3), and Series 3 uses a value of seven (7). The search times with the delay incorporated contrast the search times calculated in Figure 4.3, showing the delay is proportional to the likelihood of attack.

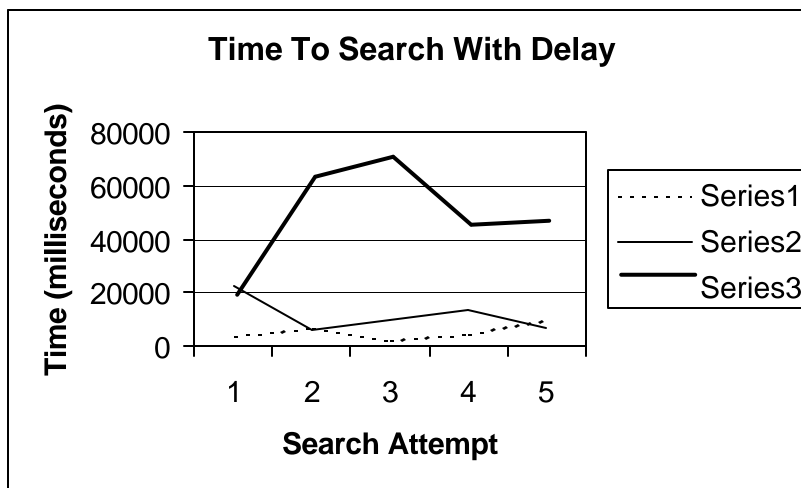

Figure 4.4 Time To Search With Delay Graph

4. Login Screen

Simulating network access is an interesting alternative deception, and it can be used to seek to confuse and entice an attacker. The appearance of a false network login screen may seem incredible and unexpected to a user, and may further confuse and entice him. We used the simulated network login screen in Figure 4.5 instead of returning the user to the input page without pictures after the delay. The result is confusing to the user,

and he may choose to try to gain access instead of actually attacking the system. Keep in mind, the login screen is simply another part of the code executing, and it has no ability to actually logon a user, but it can appear to process input. We experimented with a range of possibilities for the login screen; from allowing unlimited attempted logins with no delays to allowing a maximum of three attempts, with the third triggering the ***deathSleep*** sequence. The program can also write usernames and passwords to an archive to facilitate future recognition of the user.

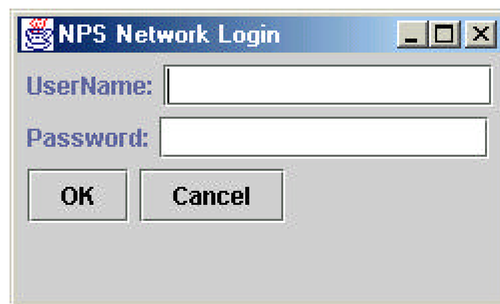

Figure 4.5 Login Screen

5. Root-Shell Simulation

The ultimate goal of an attacker is to gain root (supervisor) privileges on a computer system. Buffer-overflow attacks are often intended to produce root-shells on targeted and unprotected systems, especially those that are Unix-based. We created a fake root-shell screen that can simulate the expected results (Figure 4.6). We have two options: we can display the simulated root-shell after any buffer overflow attempt, or we can display it after some number of attempted logins. The root-shell simulation seems to be interactive to the user, but the shell provides, in fact, a scripted response generated by the software. Whatever command the user enters at the cursor, the software follows with a *"Command completed successfully"* message.

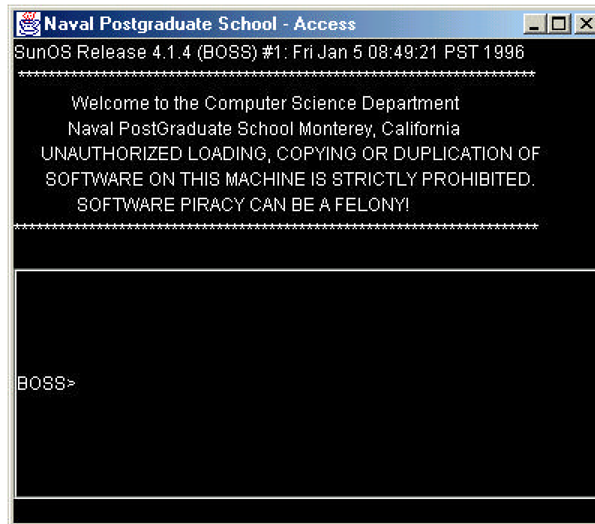

Figure 4.6 Root Shell Simulation

V. DECEPTION EXPERIMENT

A. INTRODUCTION

To better study the effectiveness of the concept of our simple deceptive response, we devised an experiment to measure the reaction of test subjects. The goals of the experiment were to determine if the subjects found the deception, whether the subjects could perceive when they were observing real or decoy-simulated delays.

1. Subjects

The participants of the study were colleagues and associates of our group. They were not provided any incentive or preparation prior to their participation. The subjects were separated into two categories: either computer student or recreational user. The subjects were not, though, potential attackers. Two subjects, however, had some experience with the study of the anatomy of computer attacks. See Table 5.1.

Table 5.1 Subjects

Subject	Background
A	Computer Student
B	Computer Student
C	Recreational User
D	Recreational User
E	Recreational User
F	Computer Student
G	Computer Student
H	Recreational User

2. Program

The program is a Java-coded program that is compiled and running on off-the-shelf personal computer with a standard software configuration. The program emulates all of the capabilities of Appendix A, while functioning without the burden of a network connection. The user is able to enter keywords into a search box, as shown in Figure 5.1 on the next page.

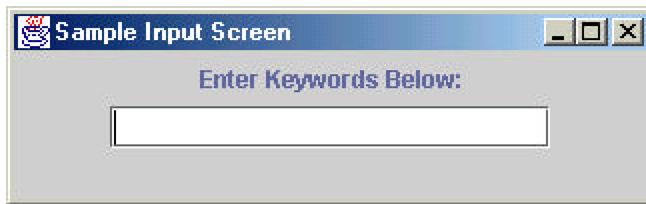

Figure 5.1 Sample Input Screen

If the keyword input is not an attack attempt, the program will respond with a message to simulate that a search was completed, and this indicates the program is operating in normal mode, as pictured below in Figure 5.2.

Figure 5.2 Message Box

However, if the user inputs keyword data that satisfies one of the trigger conditions, the ***Sleep*** method is called, and the user thread is delayed. The capability is also programmed into the software to trigger the simulated login screen and fake root-shell if a known string is entered into the keyword box.

3. Method

First, the subjects were asked two questions to establish a baseline of their perceptions. Number one: How long do they expect a keyword search program would take to provide results? Number two: If they launched an attack, how long would they think the system would take to process their malicious request?

Second, the users were individually placed in front of the computer with the program pre-compiled. They were told the keyword box would accept their input, and they were asked to type in up to five words for the program to search for. After they

completed two searches, they were told that the mode they were operating in was considered "normal."

Next, they were told to enter the string "file//." They did not know that the string would trigger the ***Sleep*** method with a *weightOfAttack* value set to 3. The value of three was chosen to ensure the delay would not be too short or too long. They were asked at this point whether they perceived if the search took longer than the two previous searches, with ***yes*** or ***no*** being their choices.

For the fourth query, they were asked to enter "#define" to trigger another delay with a higher likelihood of attack of 7, which would delay for a longer period of time. They were again asked at this point whether they perceived that their search took longer than when they were operating in normal mode, again with ***yes*** or ***no*** as choices.

The next two queries were designed to test the reaction to the presentation of the simulated login screen and the fake root-shell. We inserted two keywords that would trigger the simulations: **log** for the simulated login screen, and **boss** for the fake root-shell. The subjects were asked to enter one of these words in the search box, but not told what the system reaction would be. The subjects were then asked to give their reaction to the system response: ***surprised, not surprised, expected,*** or ***no reaction.***

Finally, the subjects were asked to provide an overall rating of the success of the deception, with the range of values being a ***1 (poor), 2 (not really believable), 3 (average believability), 4 (somewhat believable)***, and ***5 (completely believable)***. They were also asked to rate whether they were fooled or not, which we presented three possibilities: ***fooled, sort of fooled,*** and ***not fooled***.

B. RESULTS

The first two questions were asked of the subjects, and the responses were given in seconds. In some cases, the subjects were allowed to generate approximate times for their response, as shown in Table 5.2.

Table 5.2 First Two Question Answers

Subject	Search Time (sec)	Attack time to process (sec)
A	Less than 2	Approx 30
B	Less than 3	30-40
C	3	60
D	4	30
E	10	Approx 60
F	Less than 2	30-40
G	Less than 5	45 or less
H	3	Approx 30

Table 5.3 displays whether the users perceived the delays during queries three and four:

Table 5.3 Delay Perception

Subject	Perceive Delay #1?	Perceive Delay #2?
A	Yes	Yes
B	Yes	Yes
C	Yes	Yes
D	Yes	Yes
E	Yes	Yes
F	Yes	Yes
G	Yes	Yes
H	Yes	Yes

Table 5.4 now shows the reactions to the fifth and sixth queries, which sought the subject's reaction to the appearance of the simulated login screen and the fake root-shell.

Table 5.4 Reaction to Queries

Subject	Reaction to Login	Reaction to Root-Shell
A	NOT Surprised	Surprised
B	Surprised	Surprised
C	Surprised	Surprised
D	NOT Surprised	Surprised
E	Surprised	Surprised
F	Surprised	Surprised
G	Surprised	Surprised
H	Surprised	Surprised

Finally, Table 5.5 shows the subjects' comments, whether they were fooled or not, and their overall rating of the experiment.

Table 5.5 Comments and Overall Rating

Subject	Comments	Fooled?	Overall Rating
A	Believable, logical	Fooled	4
B	Very believable, effective	Fooled	5
C	Very believable, deceived	Fooled	5
D	Good job, deception should work	Fooled	5
E	Delay as expected, good graphics	Fooled	5
F	Believable, delay was what expected	Fooled	4
G	Good, delay little long, surprised	Sort of fooled	4
H	Believable, good simulation	Fooled	5

C. DISCUSSION OF RESULTS

The results of the experiment left the group optimistic about the overall validity of the simple deceptive response prototype. The test subjects had a wide range of computer-related experience, but they all reported being fooled by the deception, especially the delaying tactic. While most subjects blindly estimated the processing time before the execution of the first search, the program successfully accounted for the processing time by proving valid an earlier discussion that the perception of time is not as concrete as the perception of the actual event. That is, the subjects perceived that it took some amount of time to generate their intended reactions on the system. All of the subjects had a reasonable expectation that some event would happen during the experiment, but they seemed to have no concept of the time the program was delaying, only, in fact, that it delayed.

The simulated login screen and the fake root-shell generated better than expected reactions. Most subjects felt the appearance of the screens was surprising and believable. Only subjects A and D were not surprised by the appearance of the simulated login screen. Subject A did expect network access to become a possibility in an attack, though he stated he had some knowledge of other types of attack protocols and expected results. Subject D also had limited knowledge of attacks and hence was not completely surprised.

The root-shell simulation was not expected by any of the subjects, and successfully surprised all of the subjects. The reactions among the computer students were especially noteworthy because their subsequent interaction with the shell seemed to match their expectations for "normal" response of the system to their requests That is, when they typed in simulated commands, the "*Command completed successfully*" message provided a sufficient level of affirmation that the input was accepted.

The overall believability of the response was very high, averaging 4.6 out of a possible 5.0. The comments supported this rating, with most subjects stating the deception was either believable or, in some cases, very believable. Also, only one subject, Subject G, stated he was "sort of fooled" by the tactics used, with all of the other subjects stating they were for the most part fooled.

D. CONCLUSIONS OF EXPERIMENT

The experiment, overall, provided better than expected insight into the believability of our simple deceptive response. The subjects were all fooled by the delaying tactic, and fully surprised by the appearance of the network login screen and the fake root-shell. One of our two goals was to validate the believability of our simple deceptive response, and that goal was attained without question. The only skepticism we noted was with the more experienced subjects who thought the simulated screens were too easy to obtain. However, these subjects expressed favorable remarks at the realism of the display, especially when integrated with the delaying method. The delay tactic also provided confirming evidence for the hypothesis that potential attackers perceive the events in sequence, not explicitly the time to complete a process. The subjects were not concerned with the time it took to process their malicious request; only that it took some amount of time longer than a request took in normal (non-decoying) mode.

The second goal of the experiment was to determine if the subjects could tell they were being deceived. Since they were not told about the delay function, all subjects believed the computer was processing their malicious request, when it was actually delaying. Coupled with the high believability factor, we can conclude that the subjects

did not realize that the software application was operating in deceptive mode vice normal mode.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSION

Our goal was to evaluate three primary aspects of realizing intelligent software decoys. First, we investigated how the concept emerged out of the current state of intrusion-detection systems and their limitations. Since the concept is maturing, it was important to see how the notion developed from the current stagnant state of intrusion-detection, and the emerging cyber-warfare techniques that need to be studied and implemented.

Second, as an introduction to the concept, we expanded the definitions of the three levels of complexity a software-based decoy response should have. By defining what a simple, intermediate, or complex level of response means, we have laid the foundation for future research. As the need to protect systems continues to evolve, the levels may change, but their underlying organization should not change substantially.

Finally, we examined the technical feasibility of implementing a simple-level deceptive response for a software decoy. The simple model provided insight into the logic, technical feasibility, and practicality of generating believable software-based deceptive responses. It also magnified the need for other parts of the research to fully develop, namely the need to have well-formed abstract preconditions.

B. ACHIEVEMENTS OF RESEARCH

Our research into software-based deception fits between the capabilities of current intrusion-detection techniques and potential counter-active techniques, to better equip computer systems for the next generation of cyber-warfare. Defining the levels of response and showing how they relate to established military-deception techniques forms the basis for the future development and implementation of intelligent software decoys. Also, by demonstrating the concept is applicable to modern computer systems through the application of a simple response example, the research proved to be logically relevant and straightforward to implement. The important first step has been taken to better understand the theories involved in software-based deception.

C. LIMITATIONS OF RESEARCH

One limitation of our research was that the proof of concept was developed for one type of attack: the buffer overflow. Most intrusion-detection techniques focus on signatures and anomaly detection and are intended to be applicable to all systems. By focusing on one attack technique for this research, though, the protection is not generic and abstract. There are many possibilities of attack signatures, and this simple model does not cover all of them abstractly.

Another limitation of the research is that the techniques it protects against may not, in fact, still be valid or commonly used. The preconditions protect against one kind of attack. The decoying methods, though, can be reused for other variations of buffer-overflow attacks.

D. RECOMMENDATIONS FOR FUTURE RESEARCH

Since the technology and concept of intelligent software decoys is still in the developmental stage, there exist many avenues for research. Three particular areas are the following: development of the intermediate and complex levels of response, the concept of isolation, and the integration of the dynamic wrapping technology.

The advanced levels of response -- intermediate and complex -- should be studied and attempts should be made to develop proof-of-concept prototypes. As the simple response was developed, the idea of running canned scripts emerged as a way to implement the simulation. With intermediate-level responses, the simulation can continue to grow and protect, culminating with the complex-level response that should incorporate the supervisor concept [24] and the dynamic wrapping technology.

Second, isolating the attacker from the system is one of the main goals of the decoy. If there is any possible way to interact with the computer system directly from the simulation, the possibility will exist for further exploitation, and the results may be disastrous. Much the same way as the simulation needs to be a closely guarded secret, the interaction between a malicious user and a computer system must be strictly controlled at all times. The concept of a virtual, “escape-proof” chamber that the

simulation will run in, away from the actual operating system, should be studied and implemented. Michael and Riehle have termed this an antechamber [1].

Lastly, the dynamic wrapping technology that is being studied should be integrated with the response models. The technology involves developing an abstract language to evaluate behavior of a user within a system, and that technology could be used to protect against a large number of, if not all, attacks. The dynamic deployment capability to respond is an exciting possibility that could prove to be the best way to simulate responses.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. KEYWORD SEARCH PROGRAM WITH DECEPTION

```
package dpjulian;
//*****************************************************************/
// Servlet for user interface to the MARIE-4 system.              /
// Accepts list of keywords from the user, searches a database of  /
// rated caption candidates from RateCaps.out to find matches,     /
// lists their pages in decreasing order of rating; user can click on /
// them to go to those pages.  Modified with preconditions for input /
// of search string keywords.  Used as test to identify intrusions  /
// and run simulated reactions to intrusion.  Modified for Masters  /
// Thesis Naval Postgraduate School Summer 2002.                   /
// Author(s):  Donald P. Julian and Neil C. Rowe                   /
//*****************************************************************/
import java.util.*;
import java.io.*;
import java.text.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.lang.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.swing.*;

public class JulianSearchver2 extends HttpServlet {
    /* To hold known destemmed words of English */
  HashSet rchs = new HashSet();
    /* To hold pointers to capindex.out for a word */
  String Indexwords [] = new String[500000];
  long Indexaddresses[] = new long[500000];
    /* To store web-page, caption, and image-file index numbers for
words */
  String Word, Inputline;
  int Indexlength;
  int weightOfAttack;
  int tryChecker = 0;
  HashMap abbrevs = new HashMap(4000);

    // Initialize the servlet by loading main data structures
  public void init () {
      System.out.println("Starting init routine");
      int k, k1, k2, j, j1, j2, j3, j4, j5, jend;
    long startindex;
    String Weightstring, sourcestring, captionstring, imagestring,
SLoc,
      Abbrevline, Abbrev, Expan;
    // Load dictionary for the destemmer
    try { rowe.Destemmer.hashKnownWords(rchs);}
    catch (IOException e) {System.out.println("IO Error");}
    // Load index on words that gives caption-pictureref pairs
    k2 = 0;
    try { FileReader fr =
            new FileReader("/work/rowe/myjava/navy.capindexindex.out");
    BufferedReader br = new BufferedReader(fr);
    startindex = 0;
```

```
    while ((Inputline = br.readLine()) != null) {
      // Store for each destemmed word a byte address in index.out,
      // in alphabetical word order.
      j = Inputline.indexOf(' ');
      Word = Inputline.substring(0,j);
      SLoc = Inputline.substring(j+1,Inputline.length());
      startindex = Long.valueOf(SLoc).longValue();
      Indexwords[k2] = Word;
      Indexaddresses[k2] = startindex;
      k2++; }
    fr.close();
    FileReader fr3 = new FileReader("/work/rowe/myjava/abbrevs.txt");
    BufferedReader br3 = new BufferedReader(fr3);
    while ((Abbrevline = br3.readLine()) != null) {
      k1 = Abbrevline.indexOf(' ');
      Abbrev = Abbrevline.substring(0,k1);
      Expan = Abbrevline.substring(k1+1,Abbrevline.length());
      abbrevs.put(Abbrev,Expan); }
    fr3.close();
    FileReader fr4 = new FileReader("/work/rowe/myjava/synonyms.txt");
    BufferedReader br4 = new BufferedReader(fr4);
    while ((Abbrevline = br4.readLine()) != null) {
      k1 = Abbrevline.indexOf(' ');
      Abbrev = Abbrevline.substring(0,k1);
      Expan = Abbrevline.substring(k1+1,Abbrevline.length());
      abbrevs.put(Abbrev,Expan); }
    fr4.close(); }
    catch (IOException e) {System.out.println("File Error");}
    Indexlength = k2;
    System.out.println("navy.capindexindex.out loaded."); }

    /* Respond to a front-end request to find Web images whose captions
       match at least one of a given list of keywords. */
  public void doGet(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    /* To hold extracted set of keywords from user request, plus */
    /* index address of start of record and length of the record. */
    String KeywordArray [] = new String [100];
    long KeywordAddresses [] = new long [100];
    long KeywordBytes [] = new long [100];
    long BytesSorter [] = new long [100];
    int TreeMapMax = 10000;
    /* To hold extracted set of relevant Web pages and their captions
*/
    String stringresults [] = new String [TreeMapMax];
    /* To hold the weights for each caption result */
    double Resultweights [] = new double [TreeMapMax];
    /* To hold the index numbers for each caption result */
    long Resultnumbers [] [] = new long [TreeMapMax][3];
    /* To hold the URLs of image files */
    String Imagefiles [] = new String [TreeMapMax];
    /* To flag the caption results once shown */
    boolean Shownresults [] = new boolean [TreeMapMax];
    String keywordstring, wantpicsstring, Word, WordData, Page,
```

```
      SWeight, Sresultscount, resultline, CaptionNumbers, Webpage,
      CaptionNumberString, maxcaptioncountstring, Webpagestring,
      Captionstring, Caption, Imagelink, Pagecapstring, LastImagelink,
      Imagestring, datastring, Weightstring, tmpword;
    double Weight1, NewWeight, Weight, Weightmax, Weighttotal,
      WeightThreshold;
    Double DOldWeight, DWeight;
    int Capnum, maxcaptioncount, showableresultscount,
shownresultscount,
      k, k0, k1, k2, k3, M, Wordindex, j, j2, j5, jend,
        extraresultscount, i, i2, ilo, ihi, imed, lastimed, cmp,
        Keywordcount, Capsfound;
    Integer IWordindex;
    long startindex, Pagenumber, Captionnumber, Imagenumber,
Webpageptr,
      tmpaddr, tmpbytes, recordlength;
    Long Lstartindex;
    boolean matchflag, indexflag;
    char c;
    Collator MyCollator = Collator.getInstance();
    /* Set up servlet response */
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    System.out.println("Starting doGet");
    /* Extract the inputs from the front Web end page, SearchCaps.html
*/
    keywordstring = request.getParameter("keywords");
    wantpicsstring = request.getParameter("wantpics");
    boolean wantpics = false;
    if (wantpicsstring.equals("pics")) {wantpics = true;}
    maxcaptioncountstring = request.getParameter("maxcaptioncount");
    maxcaptioncount =
Integer.valueOf(maxcaptioncountstring).intValue();
    // HttpSession session = request.getSession(true);
    /* Use only three decimal digits of accuracy in calculation */
    DecimalFormat DF = new DecimalFormat();
    DF.setMaximumFractionDigits(3);
    DF.setMinimumFractionDigits(3);
    /* Open random-access index file */
    File af = new File("/work/rowe/myjava/navy.capindex.out");
    RandomAccessFile raf = new RandomAccessFile(af,"r");
    /* Use tree to hold weights of each relevant Web page found */
    TreeMap tm = new TreeMap();
    int KeywordCount = 0;
    int resultscount = 0;
    Weighttotal = 0.0;
    Weightmax = -1000.0;

    // Analyze the keywords and find pointers to those in the index;
    // sort by increasing size of the index record (most specific
first)
    StringTokenizer st = new StringTokenizer(keywordstring," ,;\t");
```

```
/*******************************************************************/
/****************BEGIN PRECONDITION CHECK************************/
/*******************************************************************/
/*
Here is a series of pre-conditions that need to be evaluated before the
program locates the pictures in the database.
*/
    // Local variables.
    String buffOver1 = "file//";
    String cStart = "#define";
    String slash = "//";

/*
Evaluating if the keywordstring, the input from the web page, here.  If
the string begins with 'file//', I assume an attack attempt and delay
but still return to find the pictures in the database.
*/
   if ( keywordstring.startsWith( buffOver1 ) ){
   // GO TO DELAY THEN GO TO SEARCH - LOWER PROBABILITY OF ATTACK
   // BUT DELAY THEN GET PICS
      System.out.println( "\nAttack recognized, going to sleepX1." );
      weightOfAttack = 1;
      Sleep sim = new Sleep( weightOfAttack );
   }

/*
Here, the check if for a beginning word of '#define'.  If the string
begins that way, then I assume the attacker is trying to launch code
through the browser.  If the number of words is between 1 and 20, then
I assume it may be an attack, but it could be a typo so I delay, then
search for pictures. If the length of the string is >= 20 words, then
the probability of an attack is higher, when coupled with the string
'#define'.  The value of 20 words is the chosen threshold of whether
there is following C code present or not.
*/
   if ( keywordstring.startsWith( cStart ) ){

      if ( st.countTokens() >= 1 && st.countTokens() < 20 ){
         System.out.println( "\nAttack recognized, going to sleep." );
         weightOfAttack = 2;
         Sleep sim = new Sleep( weightOfAttack );
      }
      else if ( st.countTokens() >= 20 ){
         System.out.println( "\nAttack recognized, going to DEATH
sleep." );
         weightOfAttack = 10;
         Sleep sim = new Sleep( 86 ); //Change when LogonScreen ok.
      /*   LogonScreen logon = new LogonScreen();       */
      }
   }

/* Evaluating if there are 25 or more keywords to search for.  If there
are,I assume there is a higher likelihood of attack, a of value 7.
Goes to LogonScreen routine, which it cannot recover from.
*/
```

```
    if ( st.countTokens() >= 25 ){
    //GO TO DELAY, THEN GO TO LOGON SCREEN -> DOES NOT GET PICS
       System.out.println( "\nAttack recognized, going to sleepX7." );
       weightOfAttack = 7;
       Sleep sim = new Sleep( weightOfAttack );
       LogonScreen logon = new LogonScreen();
    }

/*
Here, the evaluation is whether the keywordstring input starts with an
expression resembling a C-code program, such as a /, and whether the
length is greater than 100 characters.  If it is, then there is a very
high likelihood of attack.  I delay for a long time then proceed to the
logon screen routine to end the program.
*/
    if ( keywordstring.startsWith( slash ) ){

       if ( keywordstring.length() >= 100 ){
    // HERE, THERE IS A VERY GOOD POSSIBILITY OF AN ATTACK, SO THE
    // DELAY SHOULD BE HIGH, THEN GO TO LOGON SCREEN
            System.out.println( "\nAttack recognized, going to
            DEATHsleep." );
          weightOfAttack = 10;
          Sleep sim = new Sleep( 86 );  //Change when LogonScreen ok
      /*    LogonScreen logon = new LogonScreen();      */
       }
       else{       //The slash may be a typo, so delaying a little here
          System.out.println( "\nAttack recognized, going to sleepX1."
);
          weightOfAttack = 1;
          Sleep sim = new Sleep( weightOfAttack );
       }
    }

/*
Here, if the is only one word entered, and it has a length of greater
than 50 characters, then there is again a high probability of attack.
I delay, then proceed to the logon screen routine.
*/
    if ( st.countTokens() == 1 ){

       if ( keywordstring.length() >= 50 ){
          System.out.println( "\nAttack recognized, going to sleepX7."
);
          weightOfAttack = 7;
          Sleep sim = new Sleep( weightOfAttack );
          LogonScreen logon = new LogonScreen();
       }
    }

/*
If more than 10 words entered, calling Sleep to delay.  Sends 1 as
weight of attack for a low probability.  Delays, then continues with
finding pictures.
*/
```

```
    if( st.countTokens() >= 10 ){
       System.out.println( "\nAttack recognized, going to sleep." );
       weightOfAttack = 1;
       Sleep sim = new Sleep( weightOfAttack );
    }

    if ( keywordstring.startsWith( "Rowe" ) ){ //TEST - REMOVE LATER
       LogonScreen ls = new LogonScreen();
    }
/*******************************************************************/
/**********END OF PRECONDITIONS************************************/
/******************************************************************/

/*
/ Now, continuing with the search if not in LogonScreen.
*/
    KeywordCount = 0;
    while (st.hasMoreTokens()) {
      Word = (st.nextToken()).toLowerCase();
      if ((Word.length()>1) && (!(numberString(Word)))) {
        if (abbrevs.containsKey(Word))
            Word = ((String)abbrevs.get(Word)).toLowerCase();
      else Word = rowe.Destemmer.destem(Word,rchs);
      // Do binary lookup in main-memory index to find address
      // on disk of the record for this word
      ilo = 0;
      ihi = Indexlength;
      indexflag = true;
      lastimed = -1;
      while (indexflag) {
        imed = (ilo + ihi) / 2;
        // System.out.println("imed: " + imed + " word: " +
Indexwords[imed]);
        cmp = MyCollator.compare(Word,Indexwords[imed]);
        if (cmp == 0) {
          KeywordArray[KeywordCount] = Word;
          KeywordAddresses[KeywordCount] = Indexaddresses[imed];
          KeywordBytes[KeywordCount] =
            Indexaddresses[imed+1] - Indexaddresses[imed];
          BytesSorter[KeywordCount] = KeywordBytes[KeywordCount];
          KeywordCount++;
                indexflag = false; }


        else if (cmp<0) ihi = imed;
        else ilo = imed;
        if (imed == lastimed) indexflag = false;
        lastimed = imed; } } }
    // Interchange pairs of entries to sort keywords by record length
    Arrays.sort(BytesSorter,0,KeywordCount);
    for (i=0; i<(KeywordCount-1); i++)
      if (KeywordBytes[i] != BytesSorter[i])
      for (i2=(i+1); i2<KeywordCount; i2++)
        if (KeywordBytes[i2] == BytesSorter[i]) {
          tmpword = KeywordArray[i];
```

```
        tmpaddr = KeywordAddresses[i];
        tmpbytes = KeywordBytes[i];
        KeywordArray[i] = KeywordArray[i2];
        KeywordAddresses[i] = KeywordAddresses[i2];
        KeywordBytes[i] = KeywordBytes[i2];
        KeywordArray[i2] = tmpword;
        KeywordAddresses[i2] = tmpaddr;
        KeywordBytes[i2] = tmpbytes; }
  System.out.println("KeywordCount: " + KeywordCount);
  for (i=0; i<KeywordCount; i++)
    System.out.println(KeywordArray[i] + " " + KeywordAddresses[i] +
" " +
                 KeywordBytes[i] + " ");

  // Store pointers for each keyword into a tree
  System.out.println("Starting loop on keywords");
  Capsfound = 0;
  for (i=0; i<KeywordCount; i++) {
    Word = KeywordArray[i];
    System.out.println("Working on word '" + Word + "'");
    startindex = KeywordAddresses[i];
    recordlength = KeywordBytes[i];
    // Extract the index data from the capindex.out record
    raf.seek(startindex);
    while ((c = (char)raf.read()) != '|') {};
    c = (char)raf.read();
    while (c == ' ') {
    Weightstring = "";
    while ((c = (char)raf.read()) != ' ')
      Weightstring = Weightstring + c;
    Pagecapstring = "";
    while ((c = (char)raf.read()) != '|')
      Pagecapstring = Pagecapstring + c;
    Pagecapstring = Pagecapstring.substring(0,Pagecapstring.length()-
1);
    Weight1 = Double.valueOf(Weightstring).doubleValue();
    /* Adjust using inverse keyword document frequency. */
    Weight1 = 0.1*Weight1*Math.log(50000.0/(recordlength*0.03));
    Weighttotal = Weighttotal + Weight1;
    // Store weight found for matched caption in tree
    if (!(tm.containsKey(Pagecapstring))) {
        /* System.out.println("New page " + Pagecapstring +
           " at weight " + Weight1); */
      if (Capsfound < TreeMapMax) {
        resultscount++;
        if (Weight1 > Weightmax) Weightmax = Weight1;
        tm.put(Pagecapstring, new Double(Weight1)); } }
    else {
      /* Add the weights for each keyword mentioned in the caption */
      DOldWeight = (Double) tm.get(Pagecapstring);
      NewWeight = DOldWeight.doubleValue() + Weight1;
      /* System.out.println("Page " + Pagecapstring +
         " changed weight to " + NewWeight); */
      if (NewWeight > Weightmax) Weightmax = NewWeight;
      tm.put(Pagecapstring, new Double(NewWeight)); }
```

```
      Capsfound++;
      c = (char)raf.read(); } }
    raf.close();

    /* Now collect all the pages and fill array stringresults with
strings
       containing their weight and their link. */
    /* Set weight threshold to reduce chance of > 500 answers.  */
    double Weightaverage = Weighttotal/(double)resultscount;
    if (resultscount > 1000) WeightThreshold = Weightaverage;
    else WeightThreshold = 0.0;
    /* Build second tree to sort the Web-page results found */
    TreeMap tm2 = new TreeMap();
    Set set = tm.entrySet();
    Iterator iter = set.iterator();
    int oldresultscount = resultscount;
    resultscount = 0;
    while (iter.hasNext())  {
      Map.Entry me = (Map.Entry)iter.next();
      DWeight = (Double)me.getValue();
      Weight = DWeight.doubleValue();
      if (Weight > WeightThreshold) {
        while (tm2.containsKey(DWeight)) {
            Weight = Weight-0.0001;
            DWeight = new Double(Weight); }
        tm2.put(DWeight, me.getKey());
        resultscount++; } }
    pw.println("<HTML>\n<BODY>\n");
    if (resultscount > 0) {
      // Generate header for the returned dynamic Web page
      pw.println("<H2>Images matching keywords &quot;" + keywordstring
+
              "&quot;, in order of decreasing likelihood.  (" +
                 oldresultscount +
                 " captions matched at least one keyword.)</H2><br>");
      /* Determine the actual image file URLs and store in array */
      File afi = new File("/work/rowe/myjava/navy.capimage.out");
      RandomAccessFile rafi = new RandomAccessFile(afi,"r");
      // Find the best captions matching the keywords
      showableresultscount = Math.min(resultscount,TreeMapMax);
      System.out.println("showableresultscount: " +
showableresultscount);
      for (k=0; k<showableresultscount; k++) {
        DWeight = (Double)(tm2.lastKey());
        Weight = DWeight.doubleValue();
        resultline = (String)tm2.get(DWeight);
        tm2.remove(DWeight);
      k2 = resultline.indexOf(' ');
      k3 = resultline.indexOf(' ',k2+2);
      M = resultline.length();
      Webpagestring = resultline.substring(0,k2);
      Captionstring = resultline.substring(k2+1,k3);
      Imagestring = resultline.substring(k3+1,M);
      Pagenumber = Long.valueOf(Webpagestring).longValue();
      Captionnumber = Long.valueOf(Captionstring).longValue();
```

```
      Imagenumber = Long.valueOf(Imagestring).longValue();
      Resultweights[k] = Weight;
      Resultnumbers[k][0] = Pagenumber;
      Resultnumbers[k][1] = Captionnumber;
      Resultnumbers[k][2] = Imagenumber;
        Shownresults[k] = false;
        rafi.seek(Resultnumbers[k][2]);
        Imagefiles[k] = rafi.readLine(); }
      rafi.close();
      /* Open for random access the Web-page and caption files */
      File afs = new File("/work/rowe/myjava/navy.capsource.out");
      RandomAccessFile rafs = new RandomAccessFile(afs,"r");
      File afc = new File("/work/rowe/myjava/navy.capcaption.out");
      RandomAccessFile rafc = new RandomAccessFile(afc,"r");
      // Generate HTML for the best captions
      shownresultscount = 0;
      for (k=0; ((k<showableresultscount) &
(shownresultscount<maxcaptioncount)); k++) {
      if (!Shownresults[k]) {
          shownresultscount++;
          Imagelink = Imagefiles[k];
        Webpageptr = Resultnumbers[k][0];
        rafs.seek(Resultnumbers[k][0]);
        Webpage = rafs.readLine();
        rafc.seek(Resultnumbers[k][1]);
        Caption = rafc.readLine();
        /* If user wants pictures, insert Web image reference */
        if (wantpics) {
          pw.println("<img src='" + Imagelink +
                   "'\n alt='" + Caption + "'><br>\n");
          /* List the Web page the above picture came from */
          pw.println("The above picture is from <A HREF = '" + Webpage
+
                   "'>" + Webpage + "</A><br>\n" ); }
        else {
          /* Else list the Web page matching the keywords */
          pw.println("Try: <A HREF = '" + Webpage +
                   "'>" + Webpage + "</A><br>\n" ); }
          extraresultscount = Math.min(showableresultscount,
            1+k+(((maxcaptioncount-
shownresultscount)*(k+1))/shownresultscount) );
          /* System.out.println("shownresultscount: " +
shownresultscount +
           " extraresultscount: " + extraresultscount); */
        /* List the caption and any other captions on the same image */
        for (k2=k; k2<extraresultscount; k2++) {
            if ((!Shownresults[k2]) &
(Imagefiles[k2].equals(Imagelink))) {
              if (!(Webpageptr == Resultnumbers[k2][0])) {
                Webpageptr = Resultnumbers[k2][0];
                rafs.seek(Resultnumbers[k2][0]);
                    Webpage = rafs.readLine();
                  pw.println("The picture also appears on <A HREF = '"
+
                         Webpage + "'>" + Webpage + "</A><br>\n" ); }
```

```
            rafc.seek(Resultnumbers[k2][1]);
              Caption = rafc.readLine();
            Weight = Resultweights[k2];
            pw.println("<b>Caption of weight " + DF.format(Weight) +
                   ": </b>&quot " + Caption +  "&quot<br>\n");
            Shownresults[k2] = true; } } } }
      rafs.close();
      rafc.close(); }
    else {
        pw.println("<h3>No images matching any keywords were
found.</h3><br>\n"); }
    pw.println("</BODY>\n</HTML>\n");
    pw.close();
    // Append session info to "searchcapsscript.out"
    Date date = new Date();
    String sessiondata = date + /* " " + session.isNew() +  */
      " " + wantpicsstring + " " + maxcaptioncountstring +
      " " + resultscount + " " + keywordstring + "\n";
    byte buffer[] = sessiondata.getBytes();
    OutputStream so =
      new FileOutputStream("/work/rowe/myjava/searchcapsscript.out",
true);
    for (j=0; j<buffer.length; j++) so.write(buffer[j]);
    so.close();
    }

    /* Treat POST requests just like GET requests */
  public void doPost(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    doGet(request,response); }

  /* Says whether a string of characters represents an integer or
decimal */
  private static boolean numberString (String S) {
    boolean numberflag = false;
    int N = S.length();
    if (N > 0) {
      int i=0;
      if (S.charAt(0) == '-') i=1;
      char C;
      numberflag = true;
      while ((numberflag) & (i<N)) {
      C = S.charAt(i);
      numberflag = (((C >= '0') & (C <= '9')) | (C == '.'));
        i++; }; };
    return numberflag; }
```

```
/*********************************************************************
/ Simulated LogonScreen Class
/ When called, produces a simulated logon screen that is functionless
/ to the user.  Only escape from here is closing of connection.
/********************************************************************/
   private class LogonScreen extends JFrame{

   private FlowLayout decoy;
   private JLabel label, passName;
   private JButton ok, cancel;
   private JTextField text;
   private JPasswordField pword;

      public LogonScreen(){

         super( "NPS Network Login");

         decoy = new FlowLayout();

         Container c = getContentPane();
         c.setLayout( decoy );

         label = new JLabel( "UserName:" );
         text = new JTextField( 15 );

         passName = new JLabel( "Password:" );
         pword = new JPasswordField( 15 );

         ok = new JButton( "OK" );
         cancel = new JButton( "Cancel" );

         decoy.setAlignment( 5 );
         c.add( label );
         c.add( text );
         decoy.setAlignment( 5 );
         c.add( passName );
         c.add( pword );
         c.add( ok );
         c.add( cancel );

         cancel.addActionListener( new ActionListener(){
                public void actionPerformed( ActionEvent e ){
                   hide();
                }
         }
         );

         ok.addActionListener( new ActionListener(){
                public void actionPerformed( ActionEvent e ){
      //Allowing up to six chances to login, then putting to sleep.
                   if ( tryChecker <= 5 ){
                      Sleep loginTry = new Sleep( 1 );
                      text.setText("");
                      pword.setText("");
                      tryChecker++;
```

```
                  repaint();
               }
               else{
                  Sleep endLogin = new Sleep( 8 );
                  System.exit( 0 );
               }
            }
         }
         );

         setSize( 250, 150 );
         setLocation( 300, 150 );
         show();
      }
   }

/*****************************************************************
/ Sleep class.  Puts thread to sleep for a random time
/ when called.  Multiplied by a probability factor that is
/ passed from the preconditions.
/*****************************************************************/
   private class Sleep extends Thread{

   private int randomSleep;

   public Sleep( int degree ){

      if ( degree == 86 ) {
// If degree = 86, then passed from LogonScreen only.  Sleep for a long
time,
// then exit.  Last loop and should not be recoverable.
         try{
            Thread.sleep( 10000000 );
            System.exit( 0 );
         }
         catch( Exception excep ){
            System.out.println( "System error. Closing...." );
         }
      }
      else{
         //Calculating a random sleep time here.
         randomSleep = 1000 + (int)( Math.random() * 10000 ) ;

         try{
            Thread.sleep( degree*randomSleep );  //Thread sleeps here
            System.out.println( "\nWaking up from sleep." );
         }
         catch( Exception e ){
            System.out.println( "\nSystem error." );
         }
      }
   }
}
}
```
//END OF PROGRAM

LIST OF REFERENCES

[1] Michael, J.B. and Riehle, R.D. *Intelligent Software Decoys*. Proceedings of the Workshop, Engineering Automation for Software Intensive Systems Integration, Monterey, California, June 2001, pp. 178-187

[2] Wadlow, T.A. *The Process of Network Security.* Addison-Wesley Longman, 2000.

[3] Rowe, N., Michael, J.B., Auguston, M., Riehle, R.D. *Software Decoys for Software Counterintelligence.* Information Assurance Newsletter, June 2002, pp. 4-8.

[4] Kisiel, K.W., Rosenberg, B.F., and Townsend, R.E. DAWS: *Denial and Deception Analyst Workstation*. In Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, vol. II, IEEE Tullahoma, Tennessee, June 1989. pp 640-644.

[5] Sekar, R., Bowen, T., Segal, M. *On Preventing Intrusions by Process Behavior Monitoring*. In Proceedings of the Workshop on Intrusion Detection and Network Monitoring. The USENIX Association, April 1999, pp. 63-78.

[6] Somayaji, A. and Forrest, S. *Automated Response Using System-Call Delays*. Proceedings of the 9th USENIX Security Symposium, August 2000.

[7] Fowler, C.A. and Nesbit, R.F. *Tactical Deception in Air-Land Warfare.* Journal of Electronic Defense. June 1995.

[8] Griffith, S.B. *Sun Tzu The Art of War*. London, Oxford University Press, 1963, pp. 66-67.

[9] Bell, J.B. *Cheating and Deception*. New Brunswick, New Jersey, St. Martin's Press, 1982. pp. 45-74.

[10] Hatch, B., James, L., Kurtz, G. *Hacking Linux Exposed: Linux Security Secrets & Solutions.* Berkeley, California, Osborne/McGraw-Hill, 2001.

[11] Sundaram, A. *An Introduction to Intrusion Detection.* www.acm.org/crossroads/xrds2-4/intrus.html, July 2001.

[12] Gordeev, M. *Intrusion Detection: Techniques and Approaches.* www.infosys.tuwien.ac.at/Teaching/Courses/AK2/vor99/t13.html, June 2002.

[13] Denning, D. *Information Warfare and Security.* New York, New York, Association for Computing Machinery, Inc., 1999.

[14] Tognazzini, B. *Principles, Techniques, and Ethics of Stage Magic and Their Application to Human Interface Design.* Mountain View, California, Sunsoft, A Sun Microsystems Business, April 1993.

[15] Dietel, H., Dietel, P. *Java How to Program.* Upper Saddle River, New Jersey, Prentice-Hall, 1999.

[16] Chirillo, J. *Hack Attacks Revealed.* New York, New York, John Wiley & Sons, 2001.

[17] Katz, J. *The Poverty of Attention.* http://slashdot.org/features/01/06/28/1522228, August 2002.

[18] Cohen, F. *Simulating Cyber Attacks, Defenses, and Consequences.* http://www.all.net/journal/ntb/simulate/simulate, July 2002, Fred Cohen & Associates, March 1999.

[19] Le Poidevin, R. *The Experience and Perception of Time.* http://plato.stanford.edu/entries/time-experience, August 2002, Robin Le Poidevin, 2000.

[20] Even, L.R. *What is a Honeypot? Honey Pot Systems Explained.* http://www.sans.org/newlook/resources/IDFAQ/honeypot3.htm, July 2000, July 2002.

[21] Rowe, N. *Precise and Efficient Retrieval of Captioned Images: The MARIE Project.* http://www.cs.nps.navy.mil/research/marie/libtrends.html, August 2002.

[22] Jakarta Tomcat Server. www.javaservlethosting.com, September 2002.

[23] Fragkos, G. *An Event-Trace Language for Software Decoys*. Naval Postgraduate School, Monterey, California, September 2002.

[24] Michael, J.B., Auguston, M., Rowe, N., Riehle, R.D. *Software Decoys: Intrusion Detection and Countermeasures*. Proceedings Third Annual Workshop on Information Assurance, IEEE, West Point, New York, June 2002, pp. 130-138.

[25] Greenberg, S. *How to Structure Reports on Experiments in Human-Computer Interaction.* University of Calgary, Canada, July 2002.

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California

7. Neil Rowe
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California

8. J. Bret Michael
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California