NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN EXFILTRATION SUBVERSION DEMONSTRATION

by

Jessica Murray

June 2003

Thesis Advisor:	Cynthia E. Irvine
Second Reader:	Roger R. Schell

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** June 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Exfiltration Subversion Demonstration			**5. FUNDING NUMBERS**
6. AUTHOR(S) Jessica Murray			
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)***

A dynamic subversion attack on the Windows XP Embedded operating system is demonstrated to raise awareness in developers and consumers of the risk of subversion in commercial operating systems that may be safety critical. SCADA (Supervisory Control and Data Acquisition) systems that monitor and control our critical infrastructure depend on embedded systems.

The attack can be loaded onto a fielded system that has been subverted with a small software artifice. The artifice could be inserted into the system at any time in the system's lifecycle. The attack provides a flexible method for the attacker, who may not be the same individual who inserted the artifice, to gain total control of the subverted system. Due to the dynamic loading property of this subversion, the attacker does not have to decide the aspect of the system to be targeted until a time of her choice.

The attack does not exploit an existing flaw in the target module but is possible because the initial artifice is inserted into the kernel of an operating system where adversaries have access to source code. This thesis discusses certain aspects of known methods for developing systems free from subversion. Several projects that utilized these methods are presented.

14. SUBJECT TERMS Operating System Subversion, Computer Security, Verifiable Protection, Software wiretap			**15. NUMBER OF PAGES** 114
			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

AN EXFILTRATION SUBVERSION DEMONSTRATION

Jessica L. Murray
Civilian, Naval Postgraduate School
B.S., University of Michigan, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2003

Author: Jessica Murray

Approved by: Dr. Cynthia E. Irvine
Thesis Advisor

Dr. Roger R. Schell
Second Reader

Dr. Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A dynamic subversion attack on the Windows XP Embedded operating system is demonstrated to raise awareness in developers and consumers of the risk of subversion in commercial operating systems that may be safety critical. SCADA (Supervisory Control and Data Acquisition) systems that monitor and control our critical infrastructure depend on embedded systems.

The attack can be loaded onto a fielded system that has been subverted with a small software artifice. The artifice could be inserted into the system at any time in the system's lifecycle. The attack provides a flexible method for the attacker, who may not be the same individual who inserted the artifice, to gain total control of the subverted system. Due to the dynamic loading property of this subversion, the attacker does not have to decide the aspect of the system to be targeted until a time of her choice.

The attack does not exploit an existing flaw in the target module but is possible because the initial artifice is inserted into the kernel of an operating system where adversaries have access to source code. This thesis discusses certain aspects of known methods for developing systems free from subversion. Several projects that utilized these methods are presented.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
- A. PURPOSE....1
- B. THE SUBVERSION THREAT....2
- C. MEANS, MOTIVE AND OPPORTUNITY....3
  - 1. Means....3
  - 2. Motive....4
  - 3. Opportunity....5
- D. THE 2-CARD LOADER CONCEPT....6
- E. SUMMARY....7

II. BUILDING A SECURE SYSTEM....9
- A. INTRODUCTION....9
  - 1. Security Kernel Concept....9
  - 2. Structural Techniques for Security Kernel Based Systems....11
- B. A WELL-DEFINED INTERFACE....12
  - 1. Types of Access Control....12
  - 2. MAC Policies....12
  - 3. Security as a Safety Property....13
  - 4. MAC Models....14
- C. MINIMAL COMPLEXITY....16
  - 1. Complexity in Operating Systems....16
  - 2. Reducing Complexity....18
  - 3. Systems Minimized with Respect to a Security Policy....21
    - *a. The PDP 11/45*....21
    - *b. Kernelized Secure Operating System (KSOS)*....22
    - *c. The Multics Redesign Project*....24
    - *d. SCOMP*....24
    - *e. The NPS SASS Project*....26
    - *f. The GEMSOS Security Kernel*....28
    - *g. The VAX VMM Security Kernel*....29
    - *h. The Boeing MLS LAN*....32
- D. SUMMARY....34

III. IPSEC....37
- A. INTRODUCTION....37
- B. IPSEC ARCHITECTURE....37
  - 1. IPSec Protocols....37
    - *a. Authentication Header*....38
    - *b. Encapsulating Security Payload*....39
  - 3. IPSec Components....41
    - *a. Policy Agent*....41
    - *b. Internet Key Exchange*....41

c. Key Protection 43
d. IPSec Driver 44
4. The IPSec Process 45
C. SETTING UP IPSEC IN WINDOWS XP 45
1. IPSec User Tools 46
a. Configuring an IPSec Policy 46
b. Testing an IPSec Connection 46
c. Monitoring IPSec Connections 46
2. A Peer to Peer IPSec Connection 46
a. Configuring the IPSec Policy 47
b. Testing the Connection 50
D. SUMMARY 50

IV. WINDOWS XP EMBEDDED 51
A. INTRODUCTION 51
B. THE EMBEDDED ARCHITECTURE 51
1. Target Analyzer 53
2. Component Designer 53
3. Component Database Manager 54
4. Target Designer 54
5. Embedded Enabling Features 56
C. BUILDING AN XP EMBEDDED IMAGE 56
1. Creating an XP Embedded Image on a Hard Disk Partition 57
a. Prepare the Target Partition 57
b. Create a Customized Component for the Target Device 57
c. Import the New Component into the Component Database 57
d. Create a New Configuration in Target Designer 57
e. Build the XP Embedded Image 59
f. Boot to the XP Embedded Image 59
2. Creating an XP Embedded Image on a Bootable CD 59
a. Create an Enable Auto Layout Registry Component 59
b. Creating an El Torito Configuration 60
c. Creating the Bootable CD 61
d. Configure the Enhanced Write Filter 61
e. Create an El Torito CD 61
f. Boot From the El Torito CD 62
D. SUMMARY 62

V. AN IPSEC ATTACK 63
A. INTRODUCTION 63
B. HIGH LEVEL DESIGN 64
C. INTERFACE DESCRIPTION 64
1. User Interface 64
2. Link/Loader Interface 65
3. Artifice Base Interface 65
D. DETAILED DESIGN 65
1. The Attack 65

a. Data 65
b. Functions 66
c. Attack Package 69
2. The User Scripts 69
3. Viewing the Exfiltrated Data 71
C. FUTURE WORK 71
D. SUMMARY 71

VI. CONCLUSION 73

APPENDIX A: IMPLEMENTING THE ATTACK 75
A. INTRODUCTION 75
B. THE ASSEMBLY ENVIRONMENT 75
1. Configuring MASM and TextPad 75
2. Writing the Attack 77
3. Debugging with PEBrowse 77
4. Preparing the Scripts 77
C. THE TEST ENVIRONMENT 79
1. Using SoftICE 79
2. Executing the Attack 80

LIST OF REFERENCES 83

INITIAL DISTRIBUTION LIST 93

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. Operation of the IPSec Attack 1
Figure 2. A Kernel Based Operating System, After [AME83] 10
Figure 3. STOP Privilege Rings After [XTS99] 26
Figure 4. The MLS LAN NTCB After [BOE91] 34
Figure 5. Authentication Header From [W2KRK] 38
Figure 6. AH Transport Mode From [W2KRK] 39
Figure 7. AH Tunnel Mode From [W2KRK] 39
Figure 8. ESP Packet From [W2KRK] 40
Figure 9. ESP Transport Mode From [W2KRK] 40
Figure 10. ESP Tunnel Mode From [W2KRK] 40
Figure 11. IPSec Policy Agent From [W2KRK] 41
Figure 12. IPSec Driver From [W2KRK] 44
Figure 13. The IPSec Process From [W2KRK] 45
Figure 14. Add/Remove Snap-in 47
Figure 15. Adding the IPSec Snap-ins 48
Figure 16. Creating an IPSec Policy 49
Figure 17. Creating a new IPSec Filter 50
Figure 18. The XP Embedded Architecture 52
Figure 19. The Component Designer From [XPE2] 54
Figure 20. The Target Designer From [XPE2] 55
Figure 21. A Run Trigger Packet 70
Figure 22. A Set Trigger Packet 70
Figure 23. Configure TextPad to Build MASM From [IRVI03] 76
Figure 24. Configure TextPad to Run ASM Program From [IRVI03] 76
Figure 25. The Attack Function Template 78
Figure 26. SendIP Script for a Load Packet 79
Figure 27. Exfiltrated Module List Entry 81
Figure 28. Exfiltrated Clear Text 81

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Main Mode Protection Suite Attribute Values After [CG02] 42
Table 2. XP Embedded Components 58
Table 3. Add Component Registry Data 60
Table 4. El Torito Configuration Components 60
Table 5. The Global Data Table 66
Table 6. Find Modules Function 67
Table 7. Patching Function 68
Table 8. Activate/Deactivate Function 68
Table 9. IPSec Patch Function 69
Table 10. Useful SoftICE Commands 80

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Cynthia Irvine and Dr. Roger Schell, for their insight and the sacrifice of many red pens for my benefit. My thanks to the Microsoft XP Embedded Team for their help and the use of their source code. Thank you to Nancy, Dan, Christine, Tasha and Nell for their support. Thank you to all who made the National Science Foundation Scholarship for Service program possible for the opportunity to study at NPS.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Operating system subversion is the most sophisticated and powerful threat to computer systems and is the technique of choice for the well-funded professional attacker. Other attacks on computer systems rely on random flaws or user mistakes and are constrained to the permissions of a user account. A subversion attack inserts a trap door artifice into the most privileged area of the operating system, enabling it to bypass all security mechanisms. The trap door artifice remains dormant until triggered by the attacker. A dynamic subversion attack on the Windows XP Embedded operating system is demonstrated here to raise awareness of the risk of subversion in commercial operating systems that may be safety critical.

SCADA (Supervisory Control and Data Acquisition) systems that monitor and control our national critical infrastructure depend on embedded devices. In the future, many of these devices might run Microsoft operating systems. The means, motive and opportunity exist for an attacker to implement a subversion of our national critical infrastructure. To date, the only known way of developing systems free from subversion is through "verified protection" methods. These methods are discussed and several projects that utilized these methods are presented. There is no guarantee that the operating systems that support the national critical infrastructure and are maintaining our safety and security have not already been subverted if they have not been developed with "verified protection" methods.

The attack demonstrated here could be loaded onto a fielded system that has been subverted with a small (six lines of code) software artifice inserted into Windows XP (50 million lines of code.) The initial artifice could be inserted into the system at any time in the system's lifecycle. The attack provides a flexible method for the attacker, who may not be the same individual who inserted the artifice, to gain total control of the subverted system. Due to the dynamic loading property of this subversion, the attacker does not have to decide the aspect of the system to be targeted until a time of her choice.

For this demonstration, the encryption mechanism of IPSec (Internet Protocol Security) is bypassed to create the equivalent of a "wiretap," in which all data that is sent by the subverted machine using IPSec will be copied, sent out in the clear, and intercepted by the attacker. This is demonstrated by sending a file containing sensitive information via FTP (File Transfer Protocol) over an IPSec connection. We are not exploiting a random error, vulnerability or flaw in the Windows XP IPSec implementation, or weak cryptography, but are deliberately inserting the trap door mechanism. This attack is possible because the initial artifice is inserted into the most privileged portion of an operating system for which adversaries have access to source code directly or indirectly, e.g., via reverse engineering of a normal commercial product distribution.

I. INTRODUCTION

A. PURPOSE

Operating system subversion through the insertion of a software artifice is the most sophisticated threat to computer systems and the attack of choice for the well-funded professional. This thesis presents a demonstration of a dynamic trap door subversion of the Windows XP Embedded operating system. To demonstrate the capability of the subversion, an attack will be presented that bypasses the encryption mechanism provided by IPSec (Internet Protocol Security). To illustrate the exfiltration of clear text data, a file containing sensitive information is sent via FTP (File Transfer Protocol) across an IPSec connection (see Figure 1.) We have created the equivalent of a "wiretap" in which all data that is sent by the subverted machine using IPSec will be copied and sent out in the clear and intercepted by the attacker. We are not exploiting a random error, vulnerability or flaw in the Windows XP IPSec implementation, or weak cryptography, but are deliberately inserting the trap door mechanism.

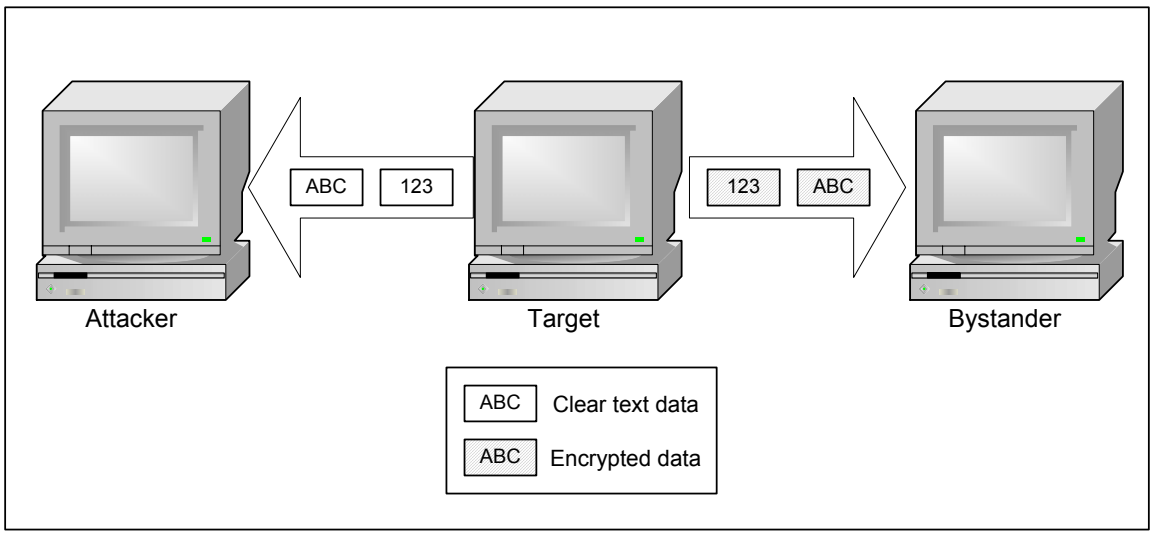

Figure 1. Operation of the IPSec Attack

Although IPSec was chosen to demonstrate the capability of the subversion artifice, this thesis also presents a strategy for attacking any kernel module. In this chapter, the technique of subversion will be compared to other computer vulnerabilities, the impact of a subversion attack on embedded systems will be discussed, and the design concept of the subversion presented here will be introduced.

B. THE SUBVERSION THREAT

Threats to a computer system can be classified as *human error*, *abuse of authority*, *direct probing*, *probing with malicious software*, *penetration*, or *subversion* [BRI95]. Attacks exploiting *human error* require determination but little skill. The attacker simply waits for a legitimate user to make a mistake that results in the disclosure of sensitive information. *Abuse of authority* attacks involve an insider abusing their authorization to violate the confidentiality, integrity or availability of the system. *Probing* attacks exploit vulnerabilities created by weak or absent security configurations and can be performed directly by the attacker or by malicious software such as Trojan horses, viruses, and worms. *Penetration* involves the exploitation of a random flaw in the system to bypass security mechanisms. *Subversion* of a system is a sophisticated attack involving the deliberate insertion of an artifice to bypass the security mechanisms of a computer system.

Subversion is defined as "…the covert and methodical undermining of internal and external controls over a system lifetime to allow unauthorized or undetected access to system resources and/or information [MYE80]." A software artifice performing subversion differs from other forms of malicious code, such as the Trojan horse. A Trojan horse has an advertised function that attracts an unsuspecting user to install the software and a covert malicious function that benefits the attacker. A Trojan horse could be disguised in music player or game software, for example, and made available for download on the web. Unlike Trojan horses, an artifice does not require any (even unwitting) action on the part of a legitimate user (such as the installation of the malicious music player or game) and is activated remotely by some triggering mechanism. An artifice remains dormant and undetectable until it is triggered. While a Trojan horse is typically confined to the permissions granted to the user who runs it, a subversion artifice runs with kernel-level permissions and is able to bypass all security mechanisms.

C. MEANS, MOTIVE AND OPPORTUNITY

The national critical infrastructure [PCCIP97], [NSPPCI] includes the energy, transportation, telecommunications, public health, water, agriculture, and shipping industries. Attacks on energy infrastructures could disrupt the oil, gas, electrical, and power systems. Subverted embedded devices used in transportation infrastructures controlling rail, air, and automotive traffic could allow trains to be diverted or collide, affect air traffic control, or cause automotive accidents through disruption of traffic signals. ATM machines could be made to export account information, or spew money spontaneously. Telecommunications could be slowed, disrupted or monitored. Public health could be affected through attacks on hospital systems and emergency response systems. Water systems, systems handling hazardous chemicals, and food production systems also utilize embedded devices and attacks could inflict significant damage. Embedded systems in weapons systems could be subverted to malfunction. Sensors used in food production and agriculture systems as well as postal and shipping infrastructures utilize embedded devices and could be vulnerable to attack.

A cyber attack could be used to amplify the effects of a physical attack on the critical infrastructures to cause total denial of service affecting the economy, national security and causing extensive disruption and casualties. Many computer systems supporting the critical infrastructure depend heavily on embedded devices and Commercial Off the Shelf (COTS) software. Supervisory Control And Data Acquisition (SCADA) systems monitoring these infrastructures utilize embedded devices [COH03]. There is no reason to believe that these systems are not subverted. This section presents recent examples illustrating that the means, motive, and opportunity [AND02] to subvert COTS systems exist.

1. Means

Subverting a system through a software artifice requires planning the insertion and design of the artifice as well as the implementation of the artifice. The attacker must have an understanding of operating systems and intermediate programming skills. Many people have these skills or could be easily trained. The artifice may vary in sophistication and skill requirements depending on what phase in the software lifecycle it is implanted,

what system is targeted, and how well it is hidden. The artifice should ideally be designed and implemented so that future legitimate updates and patches to the system will not affect its use.

As part of his Master's thesis at the Naval Postgraduate School, Emory Anderson [AND02] implemented a subversion of the Linux operating system that consisted of inserting an additional 11 lines of source code into the 4 million lines of code comprising Linux. When triggered by a malformed UDP packet specifying a user, the Network File Service (NFS) would allow that user to bypass all access control checks. This demonstration was developed under the time and resource constraints of Master's thesis work.

2. Motive

The planting of the artifice can take place well before it is ever exercised and may be implemented by one person or group and exercised by another. The actual attack may require much less skill and may allow hackers to sell the capability to interested parties. Unlike penetration attacks, which rely on the existence and discovery of an accidental vulnerability in the system, a subversion attack inserts a trap door that is guaranteed to be there.

Although obfuscation is not an objective of this research, we note that as a practical matter an artifice is virtually undetectable due to the small amount of code required and the triggering characteristic. Contemporary operating systems are complex, consisting of many interdependent modules with millions of lines of code. This complexity means that no one person can understand the system and creates the opportunity for a software artifice to go unnoticed. An artifice can also be implemented in such a way that it does not show up in the code base of the system. Thompson, one of the developers of the UNIX operating system, described during his acceptance speech for the Turing Award a way to compromise the UNIX C compiler to insert a back door into the UNIX operating system. This would allow him to log in to any UNIX system that had been compiled with the compromised compiler [THO84]. In 1974, a trap door was inserted into the Multics operating system during a security evaluation. The system was used by the Air Force Data Services Center in the Pentagon. The artifice was not found

until a year after the developers revealed which module of the system contained the artifice [KAR02][KAR74].

3. Opportunity

A system could be subverted during the design, implementation, distribution, or maintenance and support phase of its lifecycle. As an employee at a major software company, an attacker could influence the design process or implement the artifice directly. Since the few lines of code necessary for the subversion can be spread across many modules, it is unlikely that an artifice would be detected through code reviews. An artifice could also be patched into the system during distribution, or in fielded systems disguised as a legitimate patch or update. An artifice could even be introduced by a traditional penetration, e.g., exploiting a "buffer overflow" vulnerability, and persist long after patches to correct that vulnerability were installed. The artifice would not reveal itself through testing unless the trigger was guessed. A tool could be developed to systematically enter all possible inputs to a specific system with the hope of triggering an artifice. This could take months or might never be successful (it would be difficult to recognize success if it were achieved, an artifice would have to be distinguished from an ordinary flaw.) Testing can only prove the existence of flaws, not the absence of malicious code [KAR74].

Software Easter Eggs illustrate the ability of unauthorized code to slip through change management to fielded systems. These hidden features or messages that developers add to software as a signature are generally benign. One website (www.eeggs.com) lists 2771 computer Easter eggs; 149 in operating systems, 848 in applications and 134 in hardware.

Within the last few years several articles have surfaced in the news that indicate malicious individuals are aware of the powerful technique of operating system subversion and opportunities for attackers to subvert popular commercial operating systems exist. The increasing use of Commercial Off The Shelf (COTS) software in critical systems highlights the severity of these reports. In October of 2000, a hacker had access to Microsoft source code for as long as 60 days. Microsoft downplayed the damage and denied that any code had been modified. Even if the integrity of the code remained intact,

access to the source code could aid an attacker in developing an artifice added as a patch or update [KEA00]. In January of 2001, Verisign issued two Class 3 Software Publisher Certificates to an individual claiming to be a Microsoft representative. These certificates would allow an attacker to digitally sign software and distribute it as a Microsoft product [VER01]. In December of 2001, a captured Al Qaeda member told Indian police that other members of the group had infiltrated Microsoft as programmers with the intent of adding "Trojans, trapdoors, and bugs in Windows XP." Microsoft characterized the report as "bizarre" but did not deny that the individuals worked for the company [THU01]. In December of 2002, the software firm Ptech Inc., which produces enterprise data solutions used by several government agencies, was suspected of ties to Al Qaeda. Ptech source code was analyzed for the existence of subversion; not surprisingly, nothing was found, but government officials were unable to prove the absence of malicious code [VER03] [THI02].

D. THE 2-CARD LOADER CONCEPT

The subversion demonstrated in this thesis was developed in cooperation with [LACK03] and [ROG03]. The three theses demonstrate a subversion artifice modeled after the *2-card loader*. The following quote provided by [SCH03] describes the origin of the *2-card loader* concept:

> During some of my early tiger team participation with Jim Anderson and others, it was recognized that a significant aspect of the problem of Trojan horse and trap door artifices was the ability of the artifice itself to introduce code for execution. A self-contained example was a subverted complier in turn emitting an artifice, as hypothesized in the early 1970's Multics evaluation by Paul Karger and me [KAR02], which stimulated Thompson's discussion of this in his Turing lecture [THO89][THO84]. Soon after Karger's report, other tiger team members observed that the ultimately desired artifice did not have to be self-contained, but could be imported later. It was suggested that a particularly insidious packaging of this could have the initial artifice provide the functions of simple bootstrap loader typically hardwired in the computers of that era. These loaders did something like read the first two binary cards from the card reader and initiate execution of what was read, which was usually a further bootstrap program to read and execute additional binary cards. Hence this

> class of attack came to be commonly referred as the "2-card loader problem."
>
> The concept and term became quite commonplace, although I don't know of any widely reported actual implementation. Myers during his 1980 research at NPS was well aware of the 2-card loader problem, and his thesis implicitly included this in the trait of a trap door he termed "adaptability" which included being "designed to modify operating system code online." [Page 45 MYE80]. Much later Don Brinkley and I in our 1995 IEEE essay had the 2-card loader problem in mind when we briefly described a hypothetical attack where, "Among the functions built into the Trojan horse was the ability to accept covert software 'upgrades' to its own program." [Page 36 of ABR95].”

The bootstrap mechanism of the *2-card loader* is implemented by the *artifice base* developed in [LACK03] and the *link/loader* developed in [ROG03]. This thesis presents the *attack* that utilizes the services of the bootstrap to load itself onto the target system and modify a running kernel module.

E. SUMMARY

Subversion is a real threat to computer systems and a powerful tool for the professional attacker. The means, motive and opportunity exist for attackers to insert malicious artifacts into commercial operating systems. The artifice demonstrated in this thesis is based on an idea that was presented 30 years ago and was completed within 90 days of access to Windows XP source code. To date, the only way to ensure that there is no malicious code in a system is to use a combination of formal proofs and rigorous mappings to demonstrate that the behavior of the system complies with an established security policy that the system was built to enforce. The capability to do this has existed for some time and systems with such “verified protection” have been built. Since the operating systems we rely on to control our critical infrastructure and protect our sensitive information were not developed with a formal development methodology, there is no reason to believe that they do not contain subversive artifacts.

The techniques for building secure computer systems are discussed in Chapter II. The subversion presented here is demonstrated on the Windows XP Embedded operating system, the architecture of which is illustrated in Chapter III. The attack subverts the protection provided by the Windows implementation of IPSec, which is presented in

Chapter IV. The design of the developed attack is described in Chapter V and the development environment and implementation are explained in Appendix A. Conclusion and analysis of the work are in Chapter VI.

II. BUILDING A SECURE SYSTEM

A. INTRODUCTION

In order to build a secure system, it is necessary to define what it means for the system to be "secure." This characterization is documented in a security policy model that governs the development of the system. "Without models for guidance, system designers are forced to apply ad hoc security-related techniques throughout the design and implementation of a system. The model…rigorously and precisely defines the notions of 'security' and 'compromise,' and identifies elements that correspond to those in real systems [SCH75]." A system that is built to enforce a specific policy model can be proven secure with respect to this policy. It can be demonstrated that the system provides the functionality required in the policy model. To ensure that the system does not contain a subversion artifice such as the one demonstrated in this thesis, it must be shown that the system only contains functionality in support of the policy and nothing more.

1. Security Kernel Concept

A *security kernel* is "…the hardware and software that realize the reference monitor abstraction [AMES83]." A *reference monitor* is an abstraction of a system that mediates the requests of active subjects to gain access to passive objects. The reference monitor is characterized by three properties: completeness, isolation, and verifiability [AND72]. Completeness refers to the property that the reference monitor must mediate all accesses of subjects to objects. Isolation requires that the reference monitor implementation be separated from the rest of the operating system to ensure that it is "tamper proof." Verifiability means that the kernel must be small enough and structured in a way to permit formal analysis of the correspondence between the reference monitor implementation and the security policy model.

A system supporting a traditional security kernel should provide at least three execution domains of different privilege levels: one for the security kernel at the highest privilege level, one for the rest of the operating system (also referred to as the supervisor), and one for user applications at the lowest privilege level (see Figure 2.) These domains provide the isolation requirement for the reference monitor

implementation and are also known as protection rings [SHIR81] [SS72]. A *subject* in this abstraction is a process executing in a specific domain. An *object* is any entity in the system to which a subject is requesting access. Each subject and object has an access class, which the security kernel can interpret to mediate access requests according to the security policy. A *trusted subject* can operate at more than one access class to perform operations such as the downgrading of information, which the security kernel prohibits for other subjects but are necessary for the operation of the system. A *trusted path* mechanism is implemented to authenticate the kernel to the user. This mechanism, which includes a *secure attention key*, allows the user to ensure that she is communicating with the kernel and not some malicious code that is masquerading as the kernel.

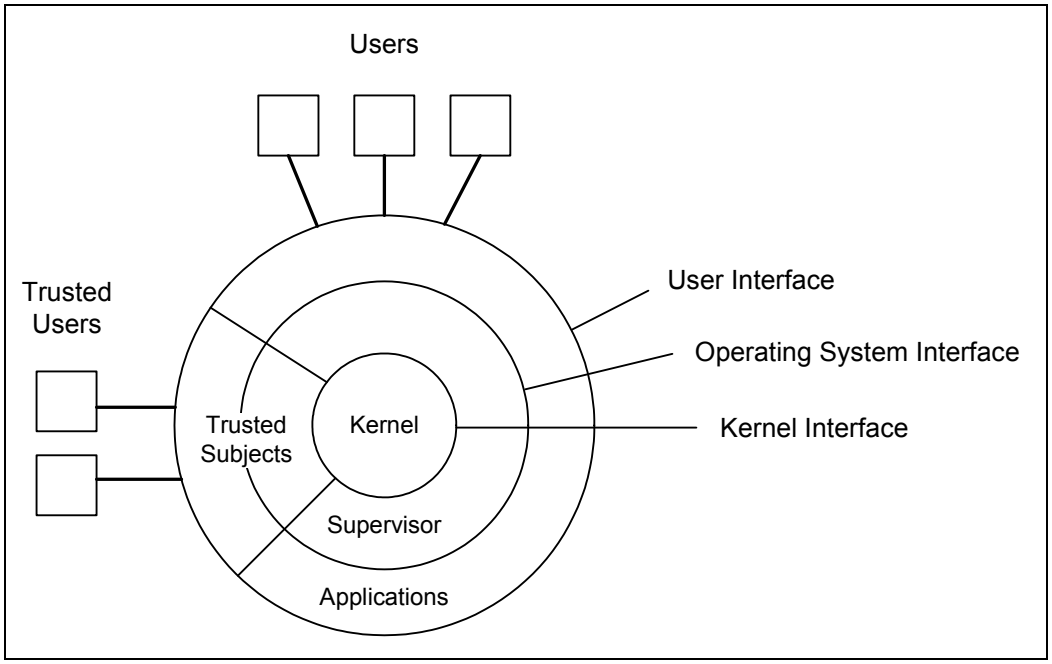

Figure 2. A Kernel Based Operating System, After [AME83]

The security kernel is a small core of security-relevant mechanisms that are specified in the security policy model. By concentrating the security-relevant mechanisms into a small, comprehensible part of the system, the system can be analyzed to verify that it enforces the security policy model in all possible system states. A simplified design is also necessary to allow the system to be analyzed for *covert*

channels. A covert channel occurs when system resources are used to signal information from a higher security level to a lower security level in violation of the security policy.

2. Structural Techniques for Security Kernel Based Systems

The techniques used to achieve the properties of isolation, completeness, and verifiability in a security kernel include: process isolation, modularity, layering, data hiding, and abstraction, effective use of hardware, principle of least privilege, a well-defined interface to the security kernel, and minimization of complexity. Each process should be isolated through the use of a separate address space. The security kernel also defines and constrains access to the address space in its own domain, which protects it from tampering. Modularity, layering, data hiding and abstraction are good software engineering practices and improve the verifiability of the system. Mechanisms implemented in hardware are generally believed to be more stable and more difficult to subvert after initial development than software and should be utilized by security functions if possible. Hardware supporting an implementation of a security kernel should include mechanisms to permit explicit processes, memory protection, execution domains, and I/O mediation [AME83]. The Principle of Least Privilege states that subjects should only be granted the minimum access level necessary to do their jobs and no more. This reduces the opportunity for misuse of the system. All functions at the user interface to the security kernel should be defined, with no undocumented functionality and all components of the security kernel identified. All of these techniques must be employed in the system to enforce a security policy and ensure that the system is free from subversion [CC99].

This chapter addresses techniques to provide the verifiability requirement through minimizing the complexity of the security kernel and the completeness requirement by defining the security kernel interface through access control policy models. A security kernel that embodies the properties of verifiability and completeness is also analyzable. Additional related properties, such as code correspondence, trusted distribution and configuration management of tools are needed to achieve verified protection. Refer to [LACK03] for an analysis of modularity, layering and abstraction and [ROG03] for an examination of the effective use of existing hardware mechanisms, which are also needed.

B. A WELL-DEFINED INTERFACE

A well-defined interface to the security kernel is required to ensure the completeness property. The interface is defined through access control policy models. There are two types of access control policies, each of which has several models that can be implemented to enforce the desired access rules.

1. Types of Access Control

Access to resources in a system can be enforced in two ways. Discretionary Access Controls (DAC), provide a run time interface for users to modify the authorizations granted to objects under their control. In a DAC system, access control lists (ACL) or Capability Lists are used to represent and implement privileges. In the first case, each file has an ACL that lists subjects and the modes (such as read, write, and/or execute) under which they are authorized to access the file using an ACL policy. In the latter case, each subject has a capability list that lists the files and access modes that are authorized using a Capability List policy. Access is granted if there is an entry allowing the requested access in the ACL or Capability list. Mandatory Access Controls (also called Non-discretionary or MAC) are used to enforce a security policy that is static, i.e., is what has been termed "global and persistent". There is no run time interface for modifying authorizations presented to users. In a MAC system all subjects and objects are characterized by a sensitivity level attribute and access is granted based on the relationship between these levels.

2. MAC Policies

In MAC policies active subjects are granted or denied authorization to access passive objects based on the sensitivity levels of the subjects and objects. A sensitivity level consists of a hierarchical classification (such as Top Secret, Secret, or Classified) and a set of non-comparable categories (such as Apples and Oranges). These categories enable the enforcement of the principle of least privilege; all those cleared for Top Secret may not need to know all the information at that level to do their jobs. A sensitivity level P is said to dominate sensitivity level R if the sensitivity classification of P is greater than or equal to the classification of R and the set of categories of R is a subset of the categories of P.

The Bell and LaPadula model is commonly used to describe a MAC policy for the confidentiality of the information protected by a system. An object can be read or modified by a subject only if two properties are satisfied: the *simple security property* and the **-property* (pronounced the "star property"). The *simple security property* states that a subject may gain read access to an object if the subject's sensitivity level dominates the object's sensitivity level. The **-property* states that a subject may gain write access to an object if the object's sensitivity level dominates the subject's sensitivity level [BLP75].

The Biba model addresses the modification or integrity of information through two properties that are the parallel of those in the Bell and LaPadula Model. The *simple security property* for integrity requires that the subject's sensitivity level must be dominated by the object's sensitivity level in order for the subject to gain read access. The integrity equivalent of the **-property* states that the sensitivity label of the subject must dominate the sensitivity level of the object [BIBA77].

These instances of MAC policies define an ideal system where information remains confined within its sensitivity domain when the system is started in a secure initial state. An implementation of the Bell and LaPadula model ensures that information from a higher confidentiality level cannot leak to a lower confidentiality level and an implementation of the Biba model ensures that information from a lower integrity level cannot contaminate information at a higher integrity level.

3. Security as a Safety Property

Harrison, Ruzzo and Ullman [HRU76] modeled the access matrix of an abstract protection system with a Turing machine to determine the safety of an arbitrary system. The safety of a system in this case is the property that there is not a series of commands that result in a subject gaining temporary access rights to an object that is contrary to the security policy. The configuration, Q, representing a protection system was defined as:

$Q = (S, O, P)$, where

$S = \{S_0, S_1 \ldots S_n\}$ is the set of current subjects

$O = \{O_0, O_1 \ldots O_n\}$ is the set of current objects

$S \subseteq O$

R is a set of access rights

P is an access matrix, such that

$P[s, o]$ = the access rights of subject s to object o, and

$P[s, o] \subseteq R$

State transitions of the Turing machine are the result of primitive *commands* (enter a right, delete a right, create a subject, create an object, delete a subject and delete an object), which are a set of *conditions* followed by *operations*. The *conditions* must be met before the *operations*, which modify the entries in the access matrix, are executed.

They proved that the safety of an arbitrary system is undecidable; it is analogous to solving the halting problem for a Turing machine. This means that there is no universal algorithm that can determine the security of an arbitrary access control policy. However, the safety of *mono-operational* systems, in which no new subject or objects can be created, is decidable. Models can be developed for specific, highly stratified systems that enable the security to be decidable with respect to a mandatory access control policy. The MAC policy enforcement of the Bell and LaPadula and the Biba models mentioned above are decidable examples. These models assume tranquility of sensitivity labels: the labels are universally consistently interpreted and do not change over time.

4. MAC Models

Denning [DEN76] introduced a Lattice Model for analyzing the information flow of a system and provided a proof that a lattice can represent the confinement properties of a mandatory access control policy. When a finite set of labels can be partially ordered and there is a least upper bound label and a greatest lower bound label, a universally bound lattice can be constructed. A least upper bound and greatest lower bound are required to allow labels composed of sensitivity levels and a set of categories to be compared with a dominance relationship. She also showed that if the labels do not form a partially ordered set, a MAC policy couldn't be represented. The ability to represent an access control policy in lattice form has the additional benefits of utilizing well-known mathematical concepts and terminology and can be efficiently represented by a computer system.

While lattice based policies are generally identified with military systems, they are flexible and can be used for other applications. Lipner concluded, "…The lattice model may in fact be applicable to commercial data processing [LIP82]." The Chinese

Wall model [BRN89] is an example of a Lattice Model policy used in the commercial sector. Careful analysis of the organizational requirements and decomposition into confidentiality and integrity sensitivity levels and categories is necessary to apply the Lattice Model to commercial systems.

Bell [BELL91] gives several examples of common policies implemented using a Universal Lattice Machine (ULM). A ULM is a lattice system with four functional extensions: binding active subjects to passive objects, exclusion of a subject from binding to more than one object at a time, the ability to roll-back to a previous system state, and n-person control or separation of privilege. The Clark & Wilson [CLW87] [SHO88] integrity model; multinational sharing for coalitions; the Chinese Wall [BRN89] secrecy model based on conflict of interest classes and used by financial institutions; and originator control models can all be expressed with a Universal Lattice Machine. Bell also argued that including policy conversion logic in the security kernel does not significantly impact complexity.

Schell and Shirley [SHIR81] introduced a technique "for evaluating the relationship between policies and mechanisms" called the *assignment technique*. This technique provides a way to evaluate the ability of a mechanism to enforce a policy by mapping (assigning) the security levels of a policy to the execution domains provided by the hardware mechanism. This implements the isolation requirement for a security kernel (also known as program integrity). The program integrity policy must include a program integrity class for each subject and object and "the ordering of the program integrity classes must be fixed according to the constraints of the policy maker." There are two properties that must be enforced to provide program integrity: the *simple program integrity condition* and the *program integrity confinement property*. The *simple program integrity condition* handles the direct threat of a subject of lower integrity modifying an executable of higher integrity and states, "if a subject has 'modify' access to an object, then the program integrity of the subject is greater than or equal to the program integrity of the object." The *program integrity confinement property* handles the indirect threat of a higher integrity subject executing a program that was modified by another subject of lower integrity and states, "if a subject has execute access to an object then the program integrity of the object is greater than or equal to the program integrity of the subject."

This technique can be used to denote a program integrity policy by assigning kernel, supervisor, utility, and user levels to rings 0 through 3 of the eight rings comprising the Multics protection ring mechanism [SS72]. Once the domains have been assigned to the mechanism (in this case the protection rings), the relationships can be analyzed to ensure that all accesses authorized by the security policy are allowed and reveal any unauthorized accesses that may be permitted. The authors concluded that the combination of a ring mechanism and security kernel design is "sufficient for enforcing computer security [SHIR81]."

Irvine and Levin [IRV01] showed that the integrity of a system is limited by the integrity of its components. A multilevel security (MLS) system that is composed of low integrity COTS (Commercial Off The Shelf) software components controlled by higher integrity multilevel management components can provide multilevel confidentiality, but can only be trusted to provide the integrity of the COTS components. COTS components are low integrity because there is no assurance that they will not modify the data in an unauthorized manner. Modified objects are labeled with the greatest lower bound of their original integrity label and the modifying component. It is important to note this when determining the level of trust to place in systems utilizing COTS components.

C. MINIMAL COMPLEXITY

This section will discuss some of the common causes of complexity in operating systems, design principles for minimized systems, and introduce several examples of systems that contributed to the field of minimized security kernel development. It is important to study the findings of past computer scientists tackling the problem of complexity so that we do not fall victim to "ignorant originality" by reinventing an existing solution that may have existing flaws [PAR96].

1. Complexity in Operating Systems

Contemporary operating systems perform many functions and are inherently large and complex. Parnas [PAR96] identified several factors contributing to the complexity and size of operating systems. "Software aging" occurs when code is incrementally modified and loses the cohesion of its original design. Compatibility support for applications written on previous versions of the operating system also inflates the code

base and adds a layer of complexity. Features added to compensate for hardware limitations and performance goals contribute to the complexity of the system as well.

In his "Plea for Lean Software" Wirth [WIR95] identifies the adoption of unnecessary features as one of the primary causes of complexity in software. Vendors should refrain from adopting any features that users request without assessing the impact of the addition on the overall system design. The monolithic design paradigm, in which all features are installed while the user may require only a few, also contributes to the complexity of systems. A system should be designed based upon an intuitive metaphor and refined over time, but developers are constantly under time pressure to be first in the market or release the latest features. Unfortunately, good engineering does "not pay off in the short run."

For a system using formal verification techniques, a formal model of the security policy is drafted. From the formal model, a Formal Top Level Specification (FTLS) of the functions and mechanisms of the security kernel that are necessary to implement the formal model of the security policy is established. A formal mathematical mapping from the FTLS to the formal security policy model and an informal mapping of the implementation of the security kernel to the FTLS must be provided. These mappings prove by transitivity that the system performs the functions described in the security policy and nothing more. Then the system is analyzed for covert channels [DOD85]. To date, this is the only way to gain assurance that the system performs its specified functions correctly and has not been subverted. Testing alone can only prove the existence of bugs, and cannot provide assurance that the system is free of additional functionality not specified in the security policy model. For a complex system code reviews alone will not detect clandestine functionality since a human reviewer will not be able to understand the system as a whole.

Security kernels must be minimized in high assurance systems in order to facilitate their verification. For secure systems, the security policy of the system drives which functional modules are included in the design of the kernel and which are implemented outside the kernel. Performance and functionality issues are weighed to determine the inclusion of non security-critical modules into the kernel [AME83].

Modules that are not protection-critical and are not depended on by modules that are protection-critical are not implemented in the security kernel. It would be impossible to verify the components of a complex system mapped to a formal security policy model.

Maureen Cheheyl, Morrie Gasser, George Huff and Jonathan Millen [CHE81] analyzed four formal verification systems: the Hierarchical Development Methodology (HDM), the Formal Development Methodology (FDM), the Gypsy verification environment, and the AFFIRM system. Each system provides a specification language processor, a verification condition generator and a theorem prover. Formal Top Level Specifications for HDM, developed by SRI International, are written in the non-procedural language SPECIAL. Formal Top Level Specifications for FDM, developed by System Development Corporation, are written in Ina Jo. Gypsy, developed at the University of Texas, shares a name with its specification language, which can also be used as a high-level programming language. AFFIRM, developed at the University of Southern California, Formal Top Level Specifications are written as a set of "algebraic axioms" that describe the behavior of the system. Several of these systems were used in the projects described in Section C.3 of this chapter.

2. Reducing Complexity

Reducing complexity is a form of art and achieved through security engineering practices and the study of existing minimized systems. When adding new features to a system, existing capabilities should be preserved and leveraged if possible and users should be given the ability to choose whether to include the new features in their configuration to avoid the "software aging" effect. Designs should show good decomposition, a hierarchical structure and well defined interfaces [PAR96]. "The security kernel approach…directly addresses the size and complexity problem by limiting the protection mechanism to a small portion of the system [AME83]."

While reducing complexity was not an explicit goal, Dijkstra's "THE" multiprogramming system introduced several design and implementation techniques that produced a small, simple system. The system utilized a strict hierarchy, synchronization with semaphores, and sequential processes. This structure allowed him to limit the number of relevant test cases necessary to put the system into every state. Starting with

the lowest layer, each layer was tested and shown to be correct before the next was added. Dijkstra was able to prove that his system performed correctly through this proof by demonstration [DIJ68].

Parnas [PAR72] decomposes all functions into two classes: those that cause the system to enter a new state and those that simply return the current state. In this way a system module can be viewed as an interface with a set of inputs and outputs and the interactions of the modules analyzed. The possible values, initial values, parameters, and effects of each function are specified. From these specifications a set of relevant properties of the system can be determined and proven correct in terms of the system security policy.

Saltzer and Schroeder [SS72] identified eight design principles for a verifiable system: economy of mechanism, complete mediation, least privilege, separation of privilege, fail-safe defaults, open design, usability, and least common mechanism. These are more actionable elaborations of the three reference monitor principles defined by [AND72]. All of these principles provide support for developing a minimized high assurance system free from subversion.

Economy of mechanism: A security kernel that is too complex for inspection will create the opportunity for errors in design and implementation and allow unintended functionality to go undetected. The principle of *economy of mechanism* directly impacts the size and complexity, and hence verifiability, of the security kernel.

Complete Mediation: The kernel should be as small as possible but also complete. All requests to access all objects, which include access resulting from normal operation, system initialization, recovery, shutdown and maintenance, must be mediated by the security kernel. In order to mediate an access request there must be a consistently applied mechanism to confirm the security properties of the requesting entity. If a security kernel is minimized but does not enforce *complete mediation,* the security of the system cannot be assured.

Least Privilege: The principle of least privilege states that a user should only be provided the minimal privilege necessary for the task at hand. If penetration occurs, the damage is contained within a smaller subset of the system. A penetration of user space is

constrained by the authorizations of the user. If all users are granted the highest privilege level all penetrations will have the potential for the greatest amount of damage. The opportunity for abuse of a granted privilege is minimized and audit log analysis can be minimized if each user's authority is minimized. The principle of *least privilege* is implemented in security kernels through the use of layering and abstraction, which enable the security kernel to be analyzed.

Separation of Privilege: The system should provide mechanisms to define a fine granularity of permission rights. For example, there should be separate privileges necessary to change passwords or change file access rights, instead of an omniscient "root" privilege. This principle simplifies the verification of each access request.

Fail-safe Defaults: The default response for an access request should be a denial, with the model explicitly stating which subjects are allowed to access which objects. This philosophy creates a "fail-safe" system in which errors in the design or implementation of the system result in a failure of access and will be detected without compromise to the security of the system. If a system crashes, for example, the default should be to deny access. This principle simplifies the security kernel's error recovery mechanism and enforces the property of complete mediation: if the kernel cannot properly mediate an access request, all access is denied.

Open Design: The system should not rely on "security through obscurity" and be available for evaluation by third parties (such as Common Criteria evaluations [CC99]) without compromising the protection of the system. An *open design* permits the system to be analyzed.

Usability: The system must be useable and provide an acceptable user interface. User documentation must be clear and readable. Functionality required by users typically adds a layer of complexity. However, a design that is minimized will be more easily understood and therefore useable.

Least Common Mechanism: Attention should be paid to resources and variables accessible to more than one subject. Timing or storage covert channels can be used to signal information to unauthorized subjects. Reduction of the *common mechanism*s directly reduces the complexity of the system.

3. Systems Minimized with Respect to a Security Policy

The following section reviews eight research projects and products that produced minimized security kernel systems: Schiller's PDP-11/45 Kernel [SCH75], KSOS [PERRINE], the Multics Redesign Project [SCS77], SCOMP [FRA83], the Naval Postgraduate School SASS project [SCH83], GEMSOS [SCH85], the VAX VMM kernel [KAR91], and the Boeing A1 LAN [BOE91]. Each system was designed to enforce a specific security policy to enforce verifiable protection at the Common Criteria EAL7 (TCSEC Class A1, in the division termed "Verified Protection") level. From these examples, we can see that a verifiable system is achievable, a system built from a formal policy model may be useable, and the secure system can provide acceptable performance [SCS77]. Not all of these properties were achieved in each system. The PDP-11/45 had covert channels and suggestions from the Multics Redesign Project were never implemented, for example. The VAX VMM provided suitable performance to support its own development; GEMSOS and SCOMP were commercially available systems. Regardless of the outcome of these projects, it is worthwhile to study the applications of the design principles and structural techniques discussed in Sections A.2 and B.2 of this chapter because verified protection is the only known way to prevent subversive artifacts such as the one demonstrated in Chapter V.

a. The PDP 11/45

Schiller [SCH75] developed a verifiable kernel prototype on the DEC PDP-11/45 by establishing four levels of abstraction: level 0 abstracted the hardware, level 1 implemented sequential processes, level 2 provided segmented virtual memory and level 3 enforced the security model over the abstractions in levels 1 and 2. The kernel abstractions create a virtual machine environment for each user with segmented virtual memory that is organized into a hierarchical directory structure. The security policy includes mandatory access control based the Bell and LaPadula model [BLP75] and discretionary access control implemented through access control lists.

The Schiller kernel is a descriptor-based system with Active Segment Table, Process Table, Process Segments, and Memory Block Table data structures. The

possible return values, parameters and effects for each kernel function are specified in a similar method to [PAR72].

b. Kernelized Secure Operating System (KSOS)

The Kernelized Secure Operating System (KSOS) [PERRINE] was designed from the start to enforce a multi-level security policy and was intended to be a provably secure UNIX replacement. The KSOS architecture was divided into 3 functional areas: a security kernel, the Non-Kernel-Security-Related (NKSR) software, and the Kernel Interface Package (KIP.) The security kernel is a minimized and complete operating system that supports a secure execution environment. The kernel was specified in SPECIAL and formally verified using an automated tool implementing the Hierarchical Design Methodology (HDM). The NKSR provides additional security-related operating system functions that execute outside the security kernel. Since KSOS was meant to replace UNIX, the KIP was implemented to support a run-time environment for UNIX applications by translating UNIX system calls into KSOS kernel calls.

The KSOS security policy included three policy models: mandatory access control was described by the Bell-LaPadula model [BLP75] for confidentiality and the Biba model for integrity [BIBA77], and discretionary access control was described by the UNIX discretionary model. The discretionary permissions of read, write and execute can be granted to the owner, a group, or all users. The kernel attaches a sensitivity label to all subjects and objects when they are created. The labels consist of a security level, a set of security categories, an integrity level, and a set of integrity categories. The rules of all three models must be satisfied for each authorization that is granted.

The kernel is divided into four modules: Process Management, Memory Management, Input/Output, and the Reference Monitor implementation. All objects in the KSOS security kernel are given a unique descriptor called a Secure Entity IDentifier (SEID), which must be passed to the kernel interface in order to access kernel objects. The Process Management module handles the creation, deletion, communication and scheduling of processes. Trusted processes, which are authorized to act outside of the security policy for special security operations such as the downgrading of information, are also supported by the Process Management component. The Memory Management

module manages the allocation, deallocation, swapping, and access control of primary memory. Memory is divided into segments that can exist in either of the Kernel, Supervisor, or User execution domains supported by the hardware. KSOS segmentation supports shared segments between processes, which can provide high bandwidth kernel mediated communication between processes. The I/O Management module handles devices, disk extents, files and file subtypes. Each device has range of a minimum and maximum security classification level for the information that it can handle. Terminal devices can have several "virtual paths" which can have different classifications. The "secure path" provides a trusted path from the user to the kernel.

The KSOS NKSR (Non-Kernel Security Related) software consists of four modules: Secure User Services, System Operation Services, System Maintenance Services and System Administration Services. The Secure User Services module initializes the system, creates and destroys user environments, manages authentication of users to the kernel through login and authentication of the kernel to the user through a trusted path. System Operation Services support printing, mounting of file systems, network connections, and a UNIX Directory Manager (UDM). System Maintenance Services include a Storage Consistency Check, Directory Consistency Check, and File System Dump/Restore that provide file system maintenance. The System Administration Services include User Registration and Removal, System Profile and Maintenance and Audit Capture Process that support the administration of a multilevel secure system.

The KSOS KIP (Kernel Interface Package) is a set of functions that implement the system calls provided by UNIX. Applications written for UNIX can be migrated to the KSOS environment with little or no modification. Performance comparisons of applications running on UNIX, in the KIP, and natively in KSOS showed that applications written for the KSOS environment had better performance than UNIX applications running the KIP. Although the KSOS KIP provided some degree of binary compatibility with UNIX systems, applications ported to KSOS from UNIX did not perform as well as those written specifically for the KSOS environment.

c. The Multics Redesign Project

In 1974, Michael Scroeder, David Clark and Jerome Saltzer [SCS77] proposed a restructuring of the Multics operating system to address two security issues. First, Multics had been written by hundreds of programmers and was deemed too large to be verified. Second, the system was not designed to enforce a specific security policy; its mechanisms were somewhat ad hoc. They had two goals: to simplify the operating system so it would be verifiable and to implement security functions specified in a formal security policy. To determine what modules were required in the security kernel and which could be implemented outside the security kernel, type extension [JAN76] was applied to the modules. Each module had a well-defined interface, modules were analyzed for dependencies and the modules were then organized into a layered structure free from dependency loops.

Eventcounts [REED79] were used to synchronize communication between confinement layers while preserving the information flow allowed by the security model. Mutual exclusion techniques such as monitors and semaphores impose a total ordering on processes within a system while eventcounts establish a relative ordering of events. An eventcount is an integer variable with an advance primitive, which increments the eventcount, and await and read primitives, which return the value of the eventcount. A process can be assigned *signaler* privileges, which enable access to the advance primitive, or *observer* privileges, which enable access to the read and await primitives for an event count. This separation of privileges allows only authorized signaler processes to signal information to observing processes, a feature that is not available through the use of semaphores. Eventcounts can be used to avoid covert channels since the signaler does not require a reply.

d. SCOMP

The Secure Communications Processor (SCOMP) developed by Honeywell [FRA83] implemented the hardware while the SCOMP Trusted Operating Program (STOP) implemented the software portions of a reference monitor implementation. SCOMP was specifically designed to be the communication processor for the mainframe Multics redesign just described above. SCOMP was the first system to

be certified as Class A1 [DOD85] by the DoD Computer Security Evaluation Center (comparable to Common Criteria EAL7 [CC99]).

The SCOMP hardware consisted of a modified Honeywell Level 6/DPS 6 processor and a Security Protection Module (SPM). The SPM is a hardware layer between the modified processor and the rest of the system, implementing the completeness and isolation requirements of the reference monitor. The SPM uses virtual addresses and a descriptor base root (DBR) to mediate all access requests. "The DBR points to the memory and I/O descriptors for the resources available to the process." The SPM uses the DBRs for memory and I/O mediation. Since the I/O mediation is done in the SCOMP hardware, I/O device drivers do not need to execute with the highest privilege level (Ring 0) and can therefore be outside the kernel, reducing its size and complexity. The SPM was implemented mainly to improve performance by implementing mediation mechanisms in hardware. It does not require all descriptors to be pre-loaded and also includes a Virtual Memory Interface Unit (VMIU) which caches recently used memory descriptors [FRA83].

The STOP operating system consisted of three components: the security kernel, the trusted software, and the SCOMP Kernel Interface Package (SKIP) as illustrated in Figure 3. The security kernel is the reference monitor implementation and handles process management, memory management, interrupt management, and auditing. Each subject and object in the system has a unique access label consisting of security and integrity levels and category sets that is static for the life of the entity. The trusted software provides three types of services: trusted user services, trusted operation services and trusted maintenance services. Trusted user services (Ring 1) are used to initiate a processing environment for the user at a particular security level and allow the user to change her password. Trusted operation services initialize the system to a secure state and ensure that the secure state is maintained. Trusted operation services initialize devices, create the audit files, load secure processes, and allow the system operator to set the system clock, shut down the system, swap audit files, and modify devices. Trusted maintenance services are provided to the administrator to initialize, verify the consistency of, and repair a kernel file system. Database management facilities are also included to allow the maintenance of the "access authentication database, the group access

authentication database, the terminal configuration database, the security map and the mountable file system database." The SKIP provides a hierarchical file system, process control mechanisms, and I/O device support capabilities to the user. Most SKIP functions execute in Ring 2 and are mapped into the user's address space to reduce overhead, while another library of SKIP routines executes in Ring 3 [FRA83].

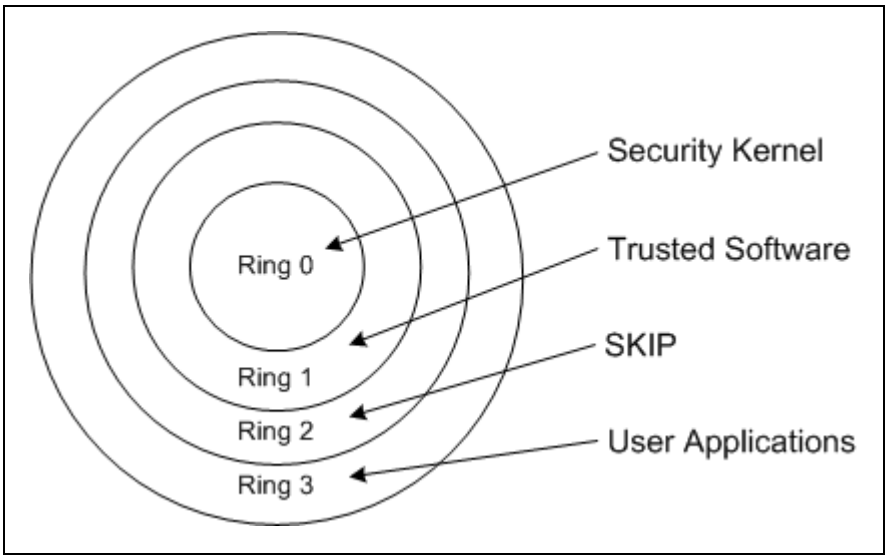

Figure 3. STOP Privilege Rings After [XTS99]

The security kernel software was formally verified through the Hierarchical Development Methodology with the FTLS written in Special [SIL83]. The trusted software was verified using the Gypsy methodology [CHE81]. Development of SCOMP was also "tightly controlled" to ensure that the implementation code corresponded with the design documentation. "The formal review by the DoD Computer Security Center was augmented by NAVELEX [the Navy Electronics Systems Engineering Center] reviews during development [FRA83]."

e. The NPS SASS Project

The Naval Postgraduate School Secure Archival Storage System (SASS) project [SCH83] was the first security kernel implemented on a commercial processor that was not expressly designed to support a security kernel. The goal of the SASS project was to illustrate the techniques for acceptable performance in a security kernel

designed for ease of verification. It should be noted that SASS was a research project for use to support graduate student instruction and to widely illustrate key engineering techniques through openly publishing its design and source code. As such, it was not intended to be a complete system that could be productized or evaluated as meeting all the requirements for verified protection. The system demonstrates the security engineering required for the construction of secure systems. The project followed a top down design and bottom up implementation approach. The system was divided into hierarchical modules providing increasing operating system capabilities. This structure allowed the modules to be developed independently and implements the design technique of information–hiding with no global data structures. Overall, "the implementation demonstrated the ability of a modern microcomputer to effectively support the security kernel approach."

There are three layers: security kernel, supervisor, and the application layer. The security kernel provides mechanisms for mandatory access control as well as management of all physical resources. The security kernel virtualized all resources, processors, storage, I/O, processors, segments, and devices. The supervisor uses these abstractions provided by the kernel to provide operating system functions, such as a file system and implements discretionary access control mechanisms. Hardware enforced execution domains separate each layer, providing isolation of the security kernel. The SASS modules are able to support several policies with only one module implementing each policy. Every subject and object is assigned a mandatory security label that is recognized by only one module in relation to the specific policy implemented by that module.

There are five layers within the security kernel: the gate keeper, segment and event managers, traffic controller, memory manager, and inner traffic controller. The gatekeeper implements the completeness property of the reference monitor, it handles any call to the kernel made by a supervisor or application level process. The segment and event managers handle all MAC access requests, and support creation and deletion of segments. The traffic controller uses event counts and sequencers to provide interprocess communications. The memory manager manages physical memory through segment descriptors and "ensures that only shared segments are in global memory" to prevent

covert channels. The inner traffic controller manages the virtual processors by providing synchronization mechanisms between virtual processors.

f. The GEMSOS Security Kernel

The GEMSOS security kernel [SCH85] [GEMSOS] was designed to meet two objectives: to implement the Class B3 requirements of the TCSEC [DOD85] and to provide a high performance security kernel. GEMSOS utilized a multiprocessor design to improve performance and a security policy model that easily represented "the security-relevant information repositories of the contemplated applications." The layered design made significant use of the Multics and SASS results identified above. Ultimately, GEMSOS was evaluated at Class A1 as a mandatory component under the Trusted Network Interpretation [DOD87].

The GEMSOS security policy model included the Bell-LaPadula confidentiality policy and the Biba integrity policy. Both secrecy and integrity labels consisted of a sensitivity level and a set of categories. Program integrity was enforced through protection rings provided by the hardware. Each subject and object had an associated ring level from 1 (highest privilege) to 7 (lowest privilege) – the 4 Intel x86 hardware privilege levels were extended to 8 fairly traditional protection rings. For a subject to gain access to an object, the object's ring level must dominate the subject's ring level.

Secondary storage in GEMSOS was divided into volumes. Each volume was composed of segments and each segment belonged to only one volume. A volume has minimum and maximum confidentiality and integrity sensitivity levels that define the minimum and maximum levels of the segments contained in the volume. Segment names were aliased to prevent covert channels. Each segment name consisted of a global name relative to a "mentor" segment and a local name that identifies the mentor segment. The secrecy and integrity levels of a segment and its mentor must follow the *compatibility* and *inverse compatibility* properties to prevent covert channels. The *compatibility* property states that the segment's secrecy level must dominate its mentor's secrecy level, while the *inverse compatibility* property states that the mentor's integrity level must dominate the

segment's integrity level. These properties ensure that an authorized process will not be barred from accessing a segment because it cannot access the segment's mentor.

To achieve the goal of a high performance security kernel, the GEMSOS kernel considered several design factors impacting performance. Security kernels are usually written in a strongly typed high-level language to support verification. Unfortunately, these languages may produce inefficient code. This is a side effect of language choice, however, and not a result of the choice to implement security. Security kernels built on hardware that lacks features for "…process management and switching, memory segmentation, Input/Output mediation, and execution domains" will require additional processor time to provide these services in software. All of these features are provided by Intel x86 processors, on which GEMSOS was built.

The use of multiprocessing in GEMSOS introduced new challenges to the development of a security kernel. Bus contention is a potential problem in multiprocessor systems, but the use of virtual, segmented memory allowed the GEMSOS kernel to determine which segments were shared and writeable (and necessary on the global bus) and which segments could be located in processor-local memory. Other systems built on a single processor typically implemented the security kernel as a single critical section. In a multiprocessor system, this introduces a degradation of service which increases as the number of processors increase. Each processor must wait while one processor finishes a call to the kernel. GEMSOS divided the kernel into several critical sections to allow significant simultaneous execution in the kernel.

g. The VAX VMM Security Kernel

Work on the VAX Virtual Machine Monitor (VMM) [KAR91] security kernel began in 1981 with five goals: to meet the Class A1 assurance requirements specified in the TCSEC [DOD85], be able to run on commercial hardware, to support existing applications, "provide acceptable performance," and be a commercially viable product. Security was one of the primary goals of the project and the kernel was built to enforce a policy including both mandatory and discretionary access controls. The layered design made significant use of the Multics and SASS results identified above.

A VMM approach was chosen for two reasons: the desire to provide support for existing software and to minimize development and maintenance costs. A VMM is not a general-purpose operating system; subjects are virtual machines and objects are virtual disks. The VMM exports an abstraction of the hardware that is "not subject to frequent change." A virtualization must address sensitive instructions, ring compression, I/O emulation, and self-virtualization. In this case sensitive means that a "privileged state of the processor" is accessed and the virtualization must trap when a sensitive instruction is called or an unauthorized subject attempts to access sensitive data. Extensions were added to the VAX architecture to allow virtualization because there were sensitive instructions and data structures that were unprotected. Since VMS uses all four VAX protection rings the protection rings were virtualized through mapping both the VM Executive and VM Kernel to the real Executive ring. No virtual machine ever runs in real Kernel mode. Extensions were also added to the VAX architecture to allow hiding of ring numbers from the VM's operating system. The compression of rings does not affect the isolation of the kernel or the VM but does reduce the robustness of the system against buggy executive mode processes. To facilitate the typically difficult endeavor of I/O emulation in VM systems, the VAX implements a "specialized call mechanism" which reduces the number of kernel traps necessary for I/O emulation. This interface required the development of a "trusted virtual device driver" for each supported operating system, a matter of only a "small number of engineer-years." "Self-virtualization is the ability of a virtual-machine monitor to run in one of its own virtual machines and recursively create second-level virtual machines." Since self-virtualization is useful mainly for developing and debugging and a user interface for self-virtualization would add complexity to the kernel, user support for self-virtualization was not implemented.

Usability was a primary goal of the VAX system but a user interface typically introduces significant complexity and is difficult to verify. To address this problem, two command sets were implemented: Secure Server Commands and SECURE commands. Secure Server commands provide a user interface to the kernel via a trusted path. The SECURE commands are the system management utilities. There are two types of SECURE commands: VM SECURE, which run in the context of the VM, and User SECURE, which are executed by the Secure Server. Facilities were implemented to

provide confirmation that an authorized user and not malicious code issued each command, the command was not modified in transit, and the command was entered into the audit log. This command confirmation was intended to reduce complexity by "eliminating need for a complex parser within the TCB" but "introduced a form of asynchronous communication… that was even more complex than a parser would have been." A menu interface and facilities for "precompiled scripts would have been simpler that the asynchronous approach."

Significant software engineering efforts were made toward reducing complexity of the VAX VMM. The developers wanted a strongly typed programming language with a quality compiler. Initially PL/I was chosen but the typing supported was not satisfactory so PASCAL was also used. However, the developers found that using two programming languages added complexity to the development process and it would have been better to stay with the original language for simplicity. Modules identified as "performance-critical" were rewritten in assembly language. The developers practiced "defensive coding" and avoided the use of global variables. Automated (DEC Module Management System) and manual techniques (visual inspection) implemented layer protection checks.

All design and code changes were reviewed against mandatory guidelines and discretionary guidelines for the consistency of design and code. Each layer was assigned a member of the development team to act as "owner" who was "responsible for the quality of that layer…[and] participated in the reviews." Interlayer problems were analyzed as well as "readability, clarity, security, performance, elegance and adherence to [design and code] guidelines." Configuration management was maintained for "design documents, trusted kernel code, test suites, user documents, and verification documents." Security reviews were used to check code against requirements and for consistency.

Robustness was purposely not included in the VAX VMM design in order to keep the complexity of the security kernel to a minimum. It was designed to be a fail-secure mechanism that would crash if any fault occurred. However, due to the "strict software engineering discipline" the system was surprisingly robust and able to support a

"heavy production load of real users" for "nearly three weeks" which was "unheard of in field test versions of brand new operating systems."

The VAX VMM had a layered design, which reduced the number of dependency loops in the system and allowed each layer to be tested separately. There are 16 layers; each layer is functionally dependent on only the abstraction of the layers below it. Event counts were used for interprocess communication. Demand paging was not implemented, which "reduces kernel complexity and improves performance at the cost of limiting the number of simultaneously active virtual machines." The system was formally specified in the Formal Development Methodology (FDM) specification language, Ina Jo, and had a TLS, FTLS, and DTLS. The VAX kernel was useable and provided adequate performance to support its own development.

h. The Boeing MLS LAN

The Boeing MLS LAN [BOE91] was developed by Boeing's Defense and Space Group and evaluated as Class A1 of the Trusted Network Interpretation [DOD87]. The security policy model of the MLS LAN includes mandatory confidentiality and integrity policies as well as a discretionary access control policy. Datagrams, connections, sessions, video circuits, and users [1] have sensitivity labels consisting of security and integrity components. Both the security and integrity components support 8 levels and 256 categories. Users have a maximum and default label; all devices have a maximum and minimum label, except for video receivers and serial connections, which have only one label. The Formal Top Level Specification of the system was written in Ina Jo and a strict configuration management plan was implemented. The system was analyzed for covert channels through informal and formal (the Flow Table Generator (FTG) developed by MITRE [MIL79]) methods. The MLS LAN includes mechanisms to support the fail-safe property.

The main component of the Boeing LAN is the Secure Network Server (SNS), which consists of a chassis with slots for up to eight processors, a system memory card, and optional video cards. The processor in the highest priority slot is the SNS processor (SNSP). The rest of the processor slots are used for Device Interface Processors

[1] One can presume that this means "processes"

(DIPs) for each user interface. There are three possible DIP types: *host DIP* for Ethernet interfaces, *serial DIP* for RS-232 serial interfaces, and *terminal DIP* for terminals. The SNS provides a serial multiplexing and terminal switch service, a serial to TELNET gateway service, a write-up connection service, an IP datagram service, inter-terminal message service, an audit server and an analog circuit-switching service.

The reference monitor abstraction is implemented in an NTCB (Network Trusted Computing Base). "The size and complexity of the NTCB has been minimized by placing the bulk of the protocol processing software (Telnet and TCP) outside the NTCB [BOE91]." The software modules of the SNS System (see Figure 4) included in the NTCB are the Executive, the Network Interface Software (NI), the SNS Management Software (SM), the Management Interface Software (MI), the Serial Communications Software (SC), the Host Communications Software (HC) and the DIP Manager Software (DM). The Executive is included on all processors of the SNS and provides services including task management and inter-task communication, segment descriptor management and processor initialization. The Network Interface Software runs on the SNSP and facilitates secure communication between SNSs in the SNS System through the enforcement of MAC (mandatory access control) on packets and trusted multiplexing, demultiplexing and addressing. The SNS Management Software executes on the SNSP and provides a Network Management Workstation locator service, an SNS initialization service, collection and distribution of audit data, SNS initialization services and MAC switching on circuit switched channels. The Management Interface Software (MI) provides an interface to the Network Management Workstation and the Audit Server. The Terminal Communications Software runs on a terminal DIP and provides inter-terminal message (ITM), Telnet sessions, and user identification and authentication services. The Serial Communications Software executes on a serial DIP and handles the establishment and termination of Telnet sessions, MAC on sessions as well as startup and shutdown of DIPs. The Host Communications Software resides on a host DIP and provides the Telnet, TCP and UDP interfaces. Each DIP includes a DIP Manager, which provides the interface between the Network Management Workstation and the Host Communication, Terminal Communication, and Serial Communication Software.

The SNS was built for the iAPX 286, which has 4 privilege levels; level 0 is the highest privilege, level 3 the lowest privilege. The ROM loader executes in level 0; the Executive executes at level 1; the Executive monitor, DIP Manager, HC, MI, NI, SC, SM, TC, and UDP execute in level 2; untrusted tasks execute in level 3.

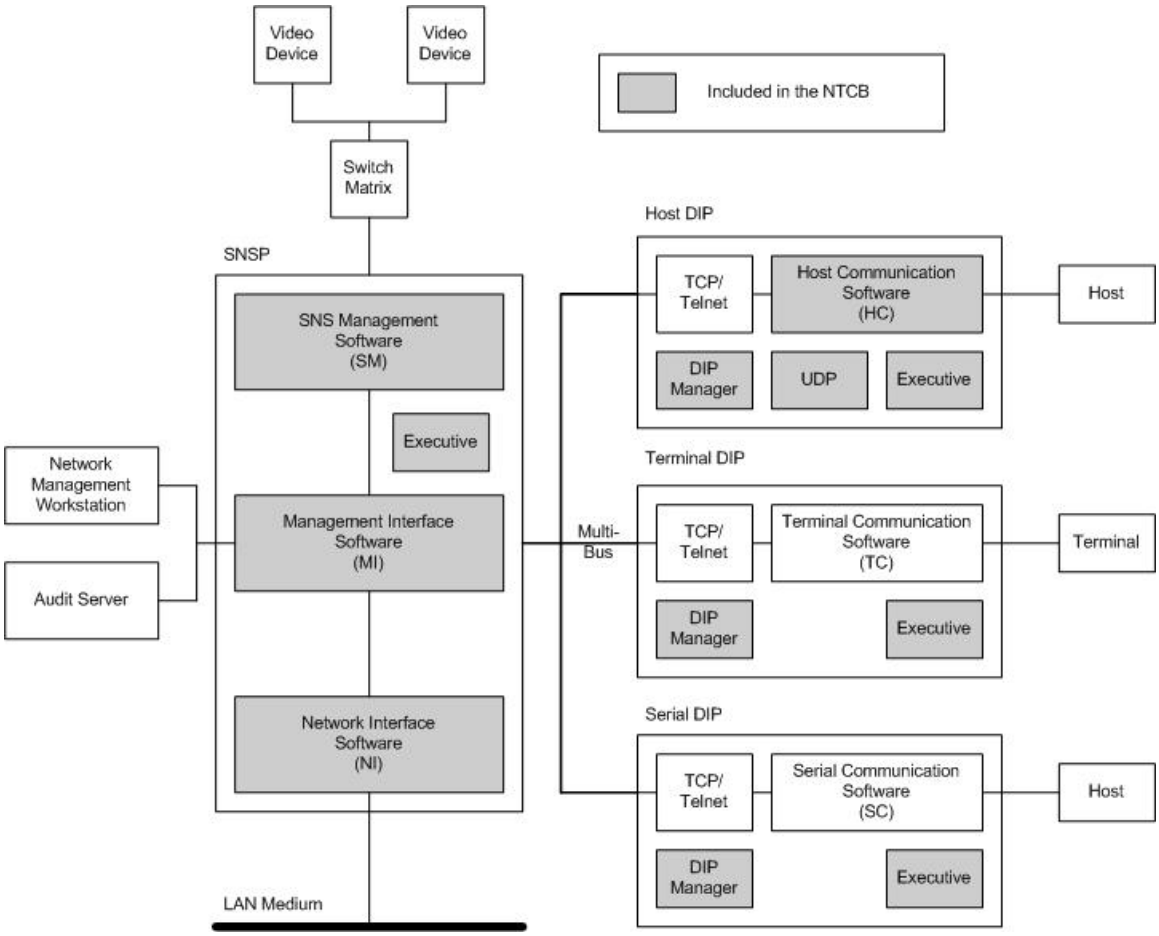

Figure 4. The MLS LAN NTCB After [BOE91]

D. SUMMARY

This chapter presented the properties, structural techniques, design principles, and several examples of projects leading to or direct examples of high assurance systems. In order to gain assurance that the system is free from subversions, such as the one demonstrated in this thesis, it must include a security kernel with verified protection based upon a security policy model and this implementation must be formally and

informally mapped to this policy. The security kernel must be complete, isolated and verifiable. These properties are achieved through process isolation, modularity, layering, data hiding, and abstraction, effective use of hardware, applying the principle of least privilege, providing a well-defined interface to the security kernel and through minimizing the complexity of the security kernel. The design should embody the principles of economy of mechanism, complete mediation, least privilege, fail-safe defaults, open design, usability, least common mechanism and separation of privilege. The remaining chapters will describe a subversion of an operating system that was possible because the system was not built to implement a specific security policy model with these properties, structural techniques, and design principles. Because the techniques for construction of a high assurance kernel were never made available to the public due to either the constraints of the research projects or the proprietary nature of the commercial efforts, a project at the Naval Postgraduate School [IRV03] is underway to construct a high assurance kernel and make all aspects of its development available.

THIS PAGE INTENTIONALLY LEFT BLANK

III. IPSEC

A. INTRODUCTION

Internet Protocol Security (IPSec) was developed by the IETF (Internet Engineering Task Force) to provide a layer of security between the Network and Transport layers of the ISO (Internet Standards Organization) OSI (Open Systems Interconnect) model [RFC2402], [RFC2406], [RFC2407], [RFC2408], [RFC2409]. The security properties provided by IPSec are: *non-repudiation*, *anti-replay*, *integrity*, *confidentiality* and *authentication*. *Non-repudiation* mechanisms allow the origin of the message to be attributed to exactly one sender. The sender cannot deny that she sent the message. *Anti-replay* mechanisms ensure that a packet is demonstrably unique. This prevents an attacker from capturing a packet and re-submitting it to gain unauthorized access to information. *Integrity* mechanisms protect data from unauthorized modification. *Confidentiality* mechanisms (encryption) protect data from unauthorized disclosure. *Authentication* mechanisms verify the identity of the message sender.

The IPSec implementation is an attractive target for attackers due to the security properties that it provides. Data that is protected by IPSec is probably sensitive and worth trying to intercept. This chapter describes the IPSec architecture and the Windows XP implementation of IPSec. Chapter V will describe an attack on IPSec in Windows XP utilizing the subversion artifice.

B. IPSEC ARCHITECTURE

1. IPSec Protocols

There are two base protocols in IPSec: Authentication Header (AH) and Encapsulating Security Payload (ESP). These protocols each provide different security properties to different parts of the packet. They can be used separately or in conjunction and in either of two modes: Transport mode or Tunnel mode. Transport mode is the more commonly used and is intended to provide protection of the upper layer protocols between two hosts. Tunnel mode protects the IP layer portion of the packet and is usually employed to protect traffic between gateways or a server and a gateway.

a. Authentication Header

The Authentication Header (AH) provides the properties of authentication, integrity, and anti-replay for the whole packet, but does not provide confidentiality. The AH is added between the Layer 4 (Transport – TCP/UDP) header and the Layer 3 (Network - IP) header (See Figure 5).

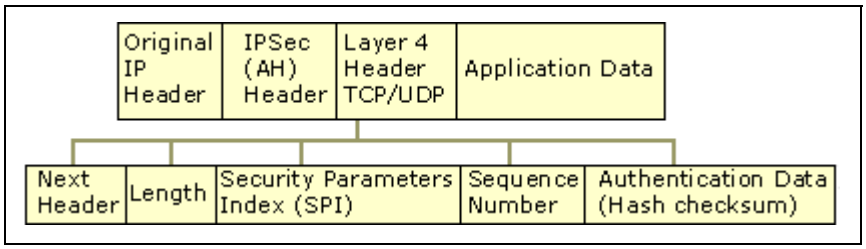

Figure 5. Authentication Header From [W2KRK]

The AH contains the fields: Next Header, Length, Security Parameters Index (SPI), Sequence Number, and Authentication Data. The Next Header is an 8-bit field indicating the type of data in the Application Data field by its protocol number defined by the Internet Assigned Number Authority (IANA) in the current Assigned Numbers database (www.iana.org.) The Length field is an 8-bit field identifying the length of the AH in 32-bit words. The Security Parameters Index (SPI) is used to determine the correct Security Association (SA -- see the Internet Key Exchange Section B.3.b). The Sequence Number is a 32-bit number that starts at 1 and monotonically increases for each packet sent with a specific Security Association. If a packet is received containing a Sequence Number that has already been received it is dropped. The Authentication Data field is a variable length field containing the Integrity Check Value (ICV), which is a hash checksum. In Transport Mode, the ICV is a checksum computed over all fields of the packet that do not vary in length (see Figure 6.)

In Tunnel Mode, the AH separates the IP addressing information into an Outer (New) IP Header and an Inner (Original) IP Header. The Original IP Header

specifies the true destination and source information and the New IP Header contains the address information of security gateways. The entire packet is signed (Figure 7).

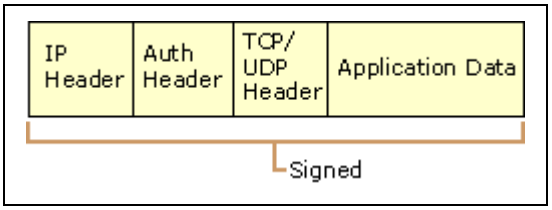

Figure 6. AH Transport Mode From [W2KRK]

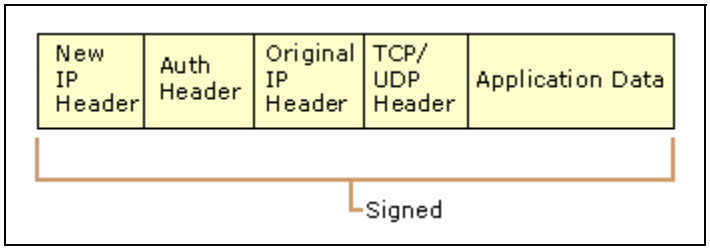

Figure 7. AH Tunnel Mode From [W2KRK]

b. Encapsulating Security Payload

The Encapsulating Security Payload (ESP) provides the properties of *confidentiality*, *authentication*, *integrity*, and *anti-replay*. The ESP inserts an ESP Header between the Layer 3 and Layer 4 headers, as well as an ESP Trailer and ESP Authentication information at the end of the IP packet (Figure 8). Unless tunneling is employed, ESP protects only the Application Data (IP payload), not the IP Header.

The ESP Header contains a Security Parameters Index and a Sequence Number identical to the AH. The ESP Trailer contains Padding, Padding Length and Next Header Fields. The Padding field of the ESP Trailer is an optional field that can vary in length from 0 to 255 bytes. It is used to align the data with the block size of the encryption algorithm. The Padding Length field specifies the length of the Padding field. The Next Header field of the ESP Trailer identifies the type of data in the Application Data field. The ESP Authentication Data is a variable length field containing an Integrity

Check Value (ICV) calculated over the ESP Header, the Application Data and the ESP Trailer, and a message authentication code (MAC) (see Figure 9).

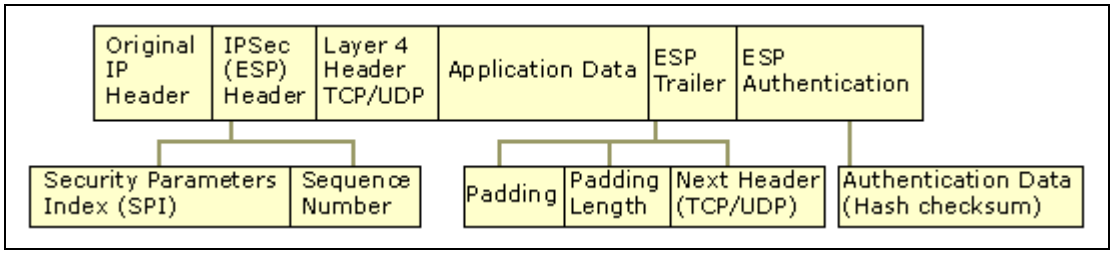

Figure 8. ESP Packet From [W2KRK]

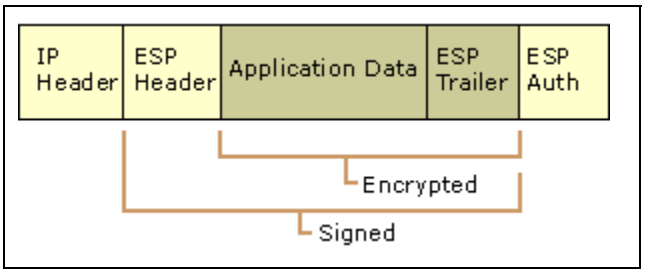

Figure 9. ESP Transport Mode From [W2KRK]

The difference between Transport and Tunnel Modes in ESP is the addition of the New IP Header and the encryption of the Original IP Header in Tunnel Mode. Unlike AH, the New IP Header in ESP Tunnel Mode is not signed (see Figure 10).

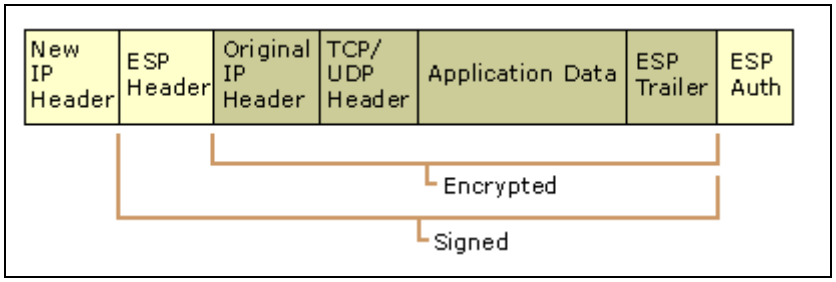

Figure 10. ESP Tunnel Mode From [W2KRK]

3. IPSec Components

The components of the Windows XP implementation of IPSec are the Policy Agent, Internet Key Exchange (IKE), Key Protection, and the IPSec Driver. These components collaborate to achieve the protection specified in the IPSec Policy.

a. Policy Agent

The Policy Agent (Figure 11) acquires the assigned IPSec Policy from the Security Policy Database (SPDB), which in Windows XP can be an Active Directory or the local registry. The Policy Agent then forwards the IP Filters specified by the assigned policy to the IPSec Driver and the authentication and encryption settings to the Internet Key Exchange component. Filter entries can specify that packets be blocked, permitted or secured based on the packet's source, destination and protocol. The Policy Agent polls the SPDB at system start time and (if the host is connected to a domain) at the default Winlogon polling interval and any interval specified in the IPSec policy.

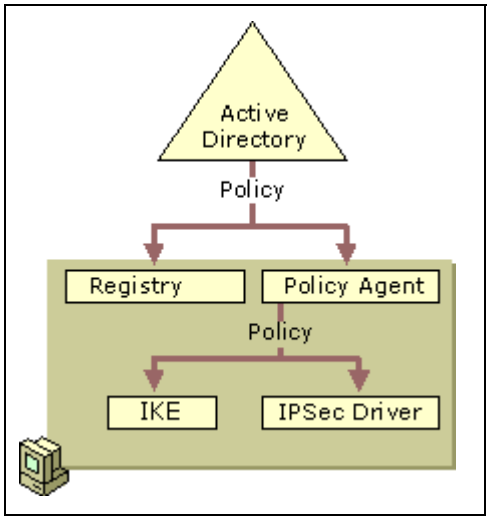

Figure 11. IPSec Policy Agent From [W2KRK]

b. Internet Key Exchange

Internet Key Exchange (IKE), a combination of the Internet Security Association Key Management Protocol (ISAKMP) and the Oakley Key Determination protocol (Oakley), is the method through which Security Associations (SAs) are

established between two communicating hosts. An SA "is a set of policy and key(s) used to protect information" [RFC2409]. In the Windows XP implementation of IPSec, IKE negotiates the SAs in two phases: Main Mode (or Phase 1) and Quick Mode (or Phase 2) [CG02].

(1) *Main Mode*. Main Mode negotiation results in an ISAKMP, or Phase 1, SA. There are three steps to Main Mode: Negotiation of the protection suite to be used, Diffie-Hellman [DIFF76] exchange of public keys, and Machine-based Authentication. Protection suites include the encryption and integrity (hash) algorithms, authentication methods, and Diffie-Hellman groups supported by each host. Microsoft Windows XP supports the protection suite attribute values specified in Table 1.

Attribute	**Attribute Value**
Encryption Algorithms	DES, 3DES
Integrity Algorithms	MD5, SHA-1
Authentication Methods	Kerberos, Preshared Key, PKI Certificate
Diffie-Hellman Groups	Group 1 (768-bit), Group 2 (1024-bit)

Table 1. Main Mode Protection Suite Attribute Values After [CG02]

The Diffie-Hellman exchange results in each communicating host holding a public key. The IKE module then uses this information to generate a shared master private key that is used to protect the authentication step and Quick Mode negotiations.

Windows XP provides three methods of machine-based authentication (the user is not authenticated through these methods): Kerberos, PKI (Public Key Infrastructure) Certificates, or Preshared Keys. Kerberos is the default and is mainly used for "client-to-server IPSec machine authentication inside the corporate network where clients and servers are members of…mutually trusted domains [CG02]." The IKE component uses the CryptoAPI to verify the certificates in PKI Certificate

authentication. Preshared Keys are stored in clear text accessible to administrators and is not recommended for a production environment [WEB3].

(2) *Quick Mode*. Quick Mode negotiation produces two IPSec, or Phase 2, SAs: one for inbound traffic and one for outbound traffic. Quick Mode negotiation is protected by the ISAKMP SA from Main Mode and is executed after Main Mode negotiation or when an IPSec SA expires [CG02]. There are three steps in Quick Mode negotiation: Policy Negotiation, Session Key Refresh or Exchange, and Distribution of the Phase 2 SAs. During the Policy Negotiation step the IPSec protocol (AH and/or ESP), the hashing algorithm (MD5 or SHA), and the encryption algorithm (DES or 3DES, if applicable) to be used are agreed upon and the two Phase 2 SAs are established. If encryption is to be used, the next step is to exchange the session keys or refresh the keying material through a Diffie-Hellman exchange. Once the SAs and keys have been established, they are distributed to the IPSec Driver together with the Security Parameter Index (SPI) of the SAs [W2KRK].

c. Key Protection

Keying material is protected in Windows XP through the following mechanisms: Key Lifetimes, Session Key Refresh Limit, Diffie-Hellman Groups, and Perfect Forward Secrecy. Key Lifetimes are specified in the IPSec policy by the administrator and can be established for both master keys and session keys. Any time a new key is generated, a new SA is also generated. Session Key Refresh Limit is enforced to protect the confidentiality of the Diffie-Hellman shared secret key, which can be degraded through reuse. When the Session Key Refresh Limit is reached, the Diffie-Hellman keys are re-negotiated. Diffie-Hellman Groups specify the length of the prime numbers used as key material in the Diffie-Hellman exchange. Windows XP supports Group 1, which protects 768 bits and Group 2, which protects 1024 bits. The larger the group, the more difficult it is to break the encryption. Perfect Forward Secrecy (PFS) ensures that keying material is only used once and can be set for the master and/or session keys. When PFS is enabled for master keys, a new Phase 1 negotiation will occur for each new Phase 2 negotiation. PFS for session keys requires less overhead since only a new Phase 2 negotiation is initiated [W2KRK].

d. IPSec Driver

The IPSec Driver (ipsec.sys) monitors all outgoing packets and compares them to the IP Filter List it received from the Policy Agent (see Figure 12.) If the filters indicate that a packet requires security, the IPSec Driver invokes the IKE component to determine the appropriate Security Associations. Once the IPSec Driver receives the outbound Phase 2 SA from the IKE component, it looks up the outbound SA in the Security Association Database (SADB) and inserts the Security Parameter Index (SPI) into the IPSec protocol header. The Driver then hashes and encrypts the appropriate fields of the packet and forwards the packet on to the IP layer to be sent out on the network.

When a packet protected by IPSec is received the IPSec Driver queries the IKE component for the session key, SA and SPI. The Driver then looks for the destination address and SPI of the SA in its SADB. The packet is then hashed to verify its integrity and decrypted if necessary. Once the packet is decoded it is forwarded to the TCP/IP driver and finally to the appropriate application.

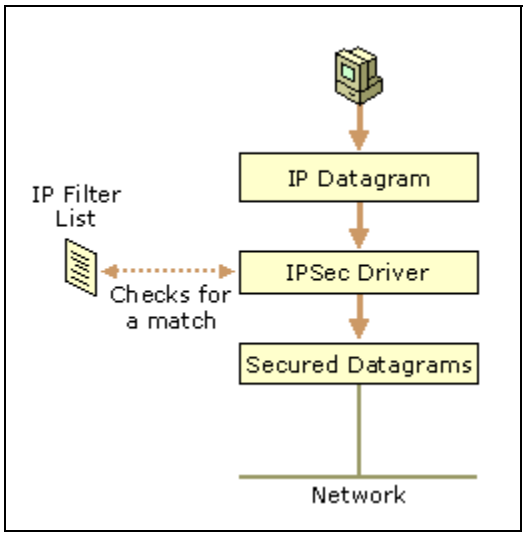

Figure 12. IPSec Driver From [W2KRK]

4. The IPSec Process

Figure 13 illustrates how the components of the Windows XP implementation of IPSec work together to provide the security properties specified in the IPSec Policy. When a user, Alissa, on Host A sends a message which matches an IPSec filter for security to a user, Brandy, on Host B, the IPSec Driver on Host A invokes the IKE component on Host A to negotiate the SAs with the IKE component on Host B. The IKE components on Host A and Host B establish a Phase I SA and a shared master key. Then the Phase II SAs are negotiated and distributed to the SA Database (SADB) used by the IPSec Drivers on each computer. The IPSec Driver on Host A then uses the newly established outbound Phase II SA to protect Alissa's message and forwards the packets to the Network layer to be sent to Brandy on Host B. When Host B's Network layer receives the packets, they are forwarded to the IPSec Driver, which verifies the security of each packet by checking the signature and decrypting the packet if necessary. The IPSec Driver then sends the packet to Host B's Transport Layer to be delivered to the application being used by Brandy. [W2KRK]

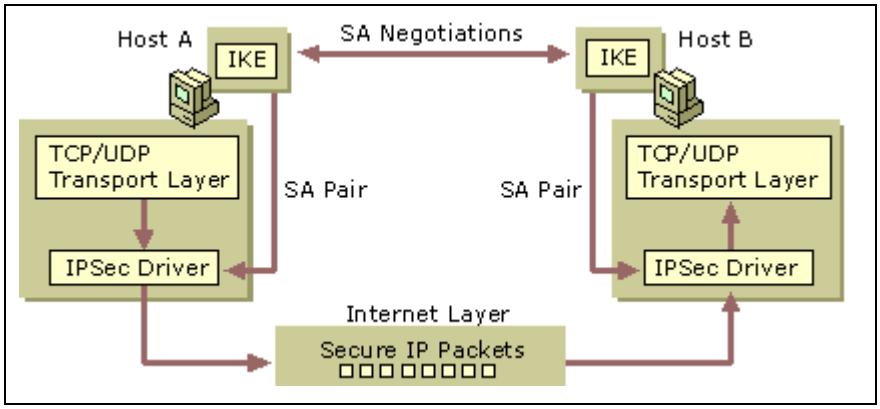

Figure 13. The IPSec Process From [W2KRK]

C. SETTING UP IPSEC IN WINDOWS XP

Now that the architecture and implementation of IPSec has been explained, we are ready to set up an IPSec connection. The tools used to configure and trouble shoot an IPSec session [WEB2] as well as step-by-step instructions for establishing the peer-to-

peer IPSec connection [CG01] used to demonstrate the IPSec attack described in Chapter V are provided in this section.

1. IPSec User Tools

a. Configuring an IPSec Policy

There are two utilities available for the configuration of IPSec policies in Windows XP: the *IPSec Security Policies snap-in for MMC* and *IPSecpol.exe*. The IPSec Security Policies snap-in for MMC is included in the Local Security Policy MMC or can be accessed from Start/Run/secpol.msc. Key Exchange settings, IP Filters, Packet Security settings, and Authentication methods can be configured and policies created, verified, exported or imported through the snap-in. IPSecpol.exe is a command line tool for creating IPSec policies instead of using the IPsec Security Policy snap-in for MMC.

b. Testing an IPSec Connection

IPSec connections can be tested with *Ping* or *Netdiag*. Ping can be used to establish an IPSec session if ICMP traffic is allowed in the policies. Netdiag supplies status information and can run diagnostic tests on networking components.

c. Monitoring IPSec Connections

The *IP Security Monitor snap-in*, *Event Viewer, IPSeccmd.exe, and Network Monitor* are used to monitor IPSec connections. The IP Security Monitor snap-in for MMC displays the Main Mode and Quick Mode SAs, and logs connection information. Some IPSec activities are saved in the Windows log files that can be viewed with the Event Viewer. Policy Agent and IPSec events are entered in the System Log; Oakley events from the IKE component events are entered in the Application Log; and ISAKMP and SA events appear in the Security Log if logon auditing is activated. IPSeccmd.exe is a command line utility that displays IPSec filter information as well as any pre-shared keys used and usage statistics. Network Monitor (Netmon) is a packet-capture tool that will decode ISAKMP, AH and ESP traffic [WEB2].

2. A Peer to Peer IPSec Connection

The first step in demonstrating a peer-to-peer IPSec connection in Windows XP [CG01] is to verify connectivity between the two hosts by pinging between them. Then the IPSec Policy can be configured on each computer and the IPSec connection tested.

a. Configuring the IPSec Policy

To configure an IPSec policy in Windows XP, first create an IPSec console containing the *IP Security Monitor Snap-in* and the *IP Security Policy Management Snap-in*. From the Microsoft Management Console (Start Run: MMC), select File: Add/Remove Snap-in (see Figure 14) and click on the Add button. Choose the IP Security Monitor and IP Security Policy Management for this computer (see Figure 15), then click Close and OK. Save this console for future use.

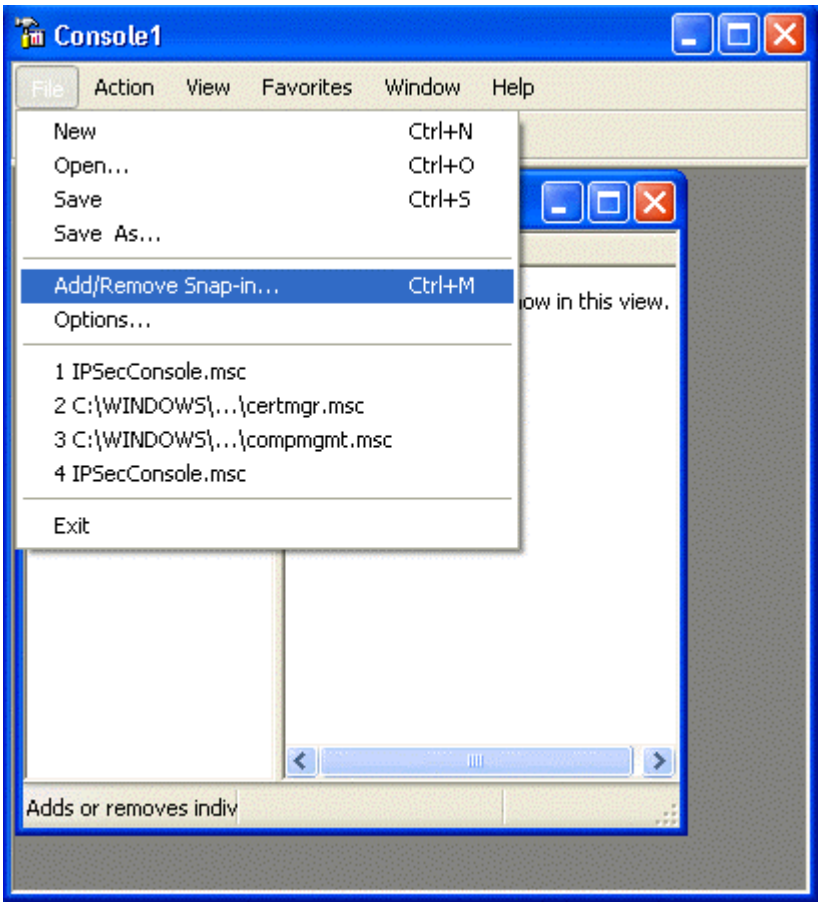

Figure 14. Add/Remove Snap-in

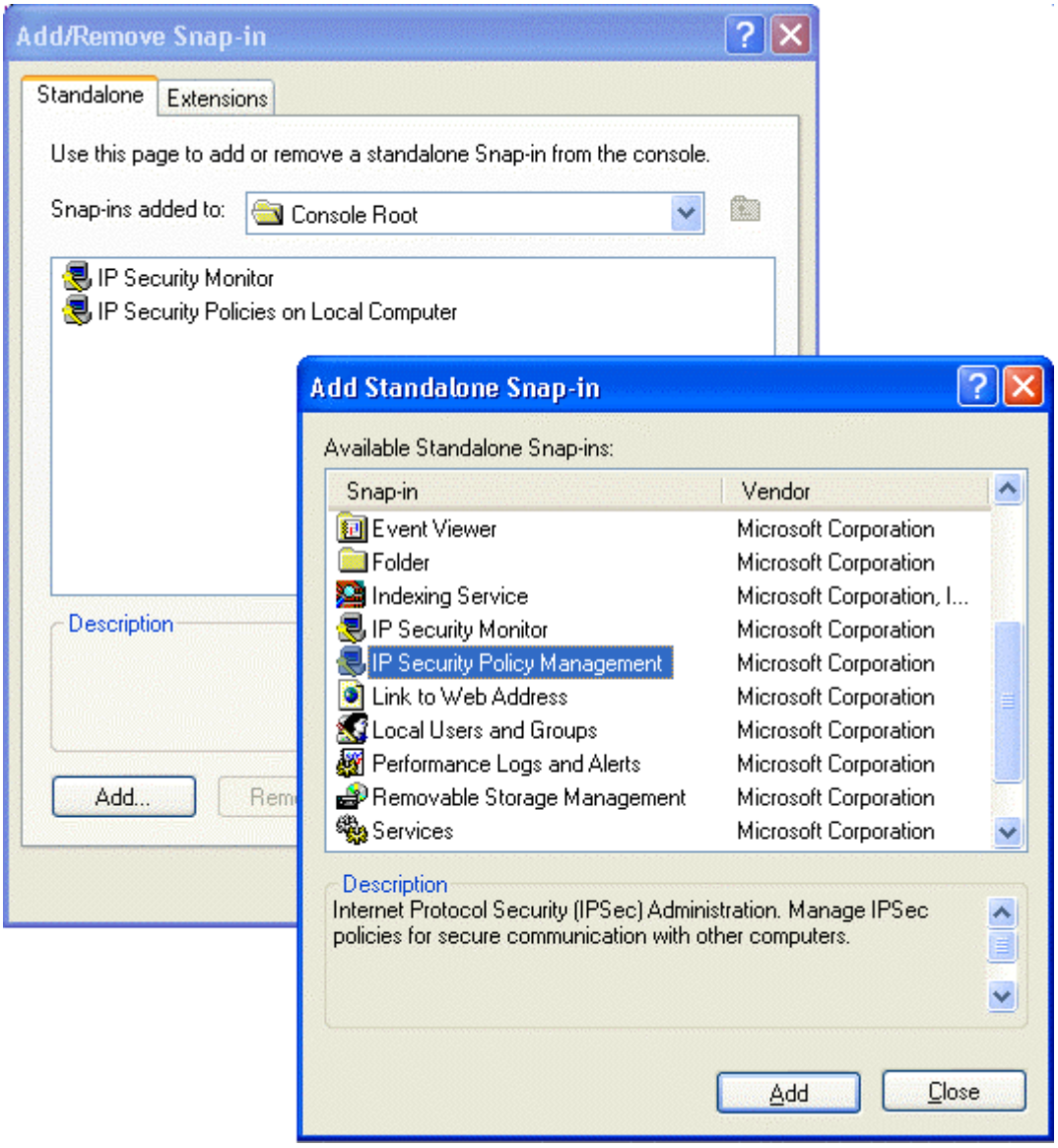

Figure 15. Adding the IPSec Snap-ins

To create a new policy, right click on the *IP Security Policies on Local Computer Snap-in* and select 'Create IP Security Policy' (see Figure 16.) Name the new policy 'Subversion' and make sure that the Activate Default Response check box is not selected in the 'Requests for Secure Communication' dialog.

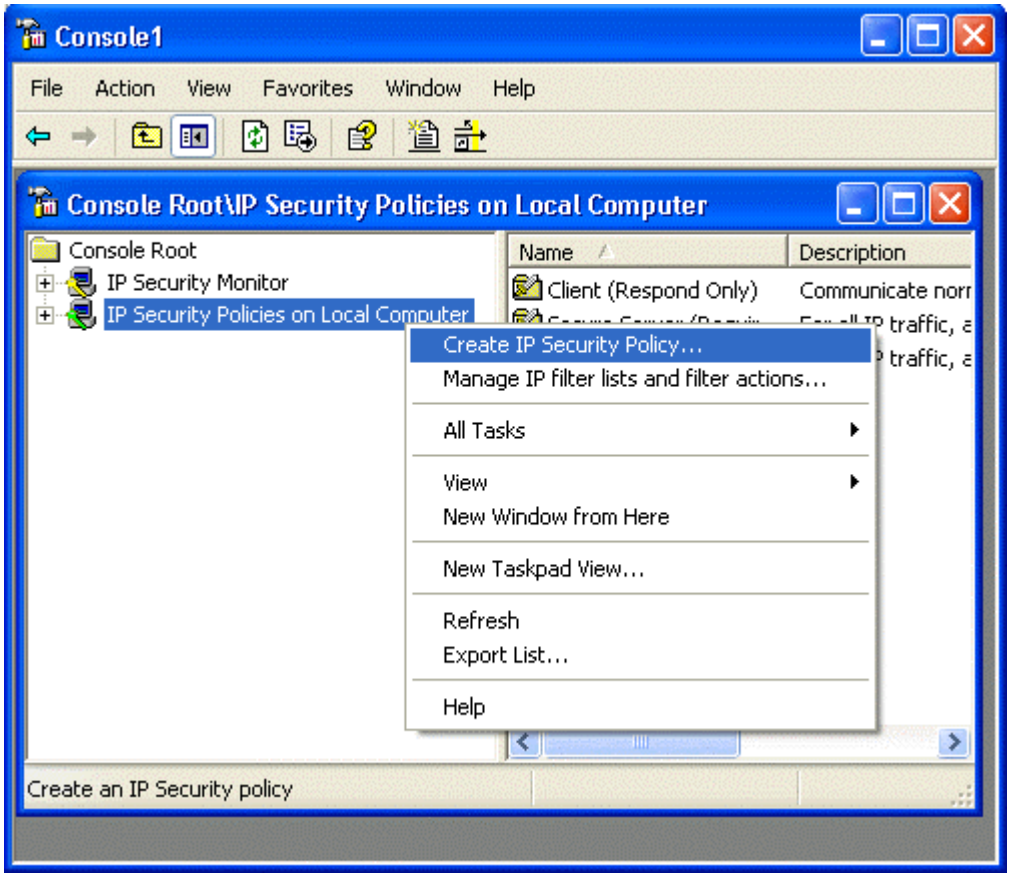

Figure 16. Creating an IPSec Policy

Right click on the 'Subversion' policy and select Properties. Click the Add button on the Rules tab of the Properties dialog box to assign a key and add a filter to the policy. Accept the defaults until you reach the Authentication Method screen. Choose 'Use this key to protect the key exchange (preshared key)' and enter 123456789 as the preshared key. Note that this is the least secure of the three authentication methods provided by the Windows XP implementation of IPSec but the attack described in this thesis does not attack the strength of the encryption keys. On the IP Filter List screen choose Add. Name the new filter 'Subversion Filter' (see Figure 17). Choose to protect all traffic from this computer to the other computer being used in the demonstration.

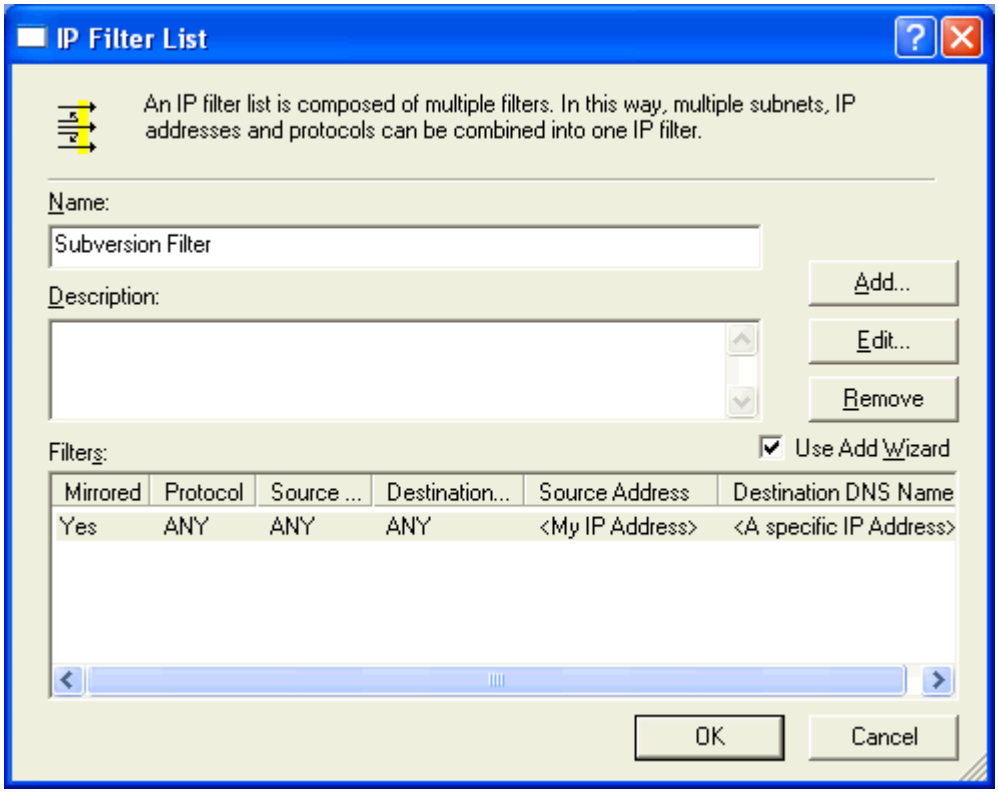

Figure 17. Creating a new IPSec Filter

b. Testing the Connection

After both hosts have been configured, test the IPSec connection by pinging between them. At first, a series of 'Negotiating IP Security' messages will be displayed; ping again and the replies should be received successfully. IPSec now protects all traffic between the two peers and the connection can be monitored with the tools described in Section C.1 of this chapter.

D. SUMMARY

This chapter presented the Windows XP implementation of IPSec and provided instructions to set up the IPSec connection used to demonstrate the attack on the IPSec Driver described in Chapter V. The next chapter will describe the Windows XP Embedded architecture and the process of creating Windows XP Embedded run-time images.

IV. WINDOWS XP EMBEDDED

A. INTRODUCTION

Although the subversion artifice described in this thesis is capable of running on Windows NT, Windows 2000 or Windows XP, it is demonstrated on a Windows XP Embedded platform. Windows XP Embedded [XPE1] [XPE2] is a "componentized" version of Windows XP. The Embedded version has the same binaries as XP, which are organized into *components*. There are over 10,000 XP operating system components available to the developer; each defines a capability that a run-time image may require in terms of resource files and properties, which are stored in a database. A Windows XP Embedded run-time image is created from a *configuration*, which is a set of components and properties. Developers can pick and choose which functionality to include in the image by selecting components to include in a configuration. To run an application on XP Embedded, it must be packaged in a component to ensure that all components that it is dependent on are also added to the configuration. Once a component has been added to a configuration, it is referred to as an *instance*. Typical configuration properties include the configuration name, author, and advanced properties such as the target boot drive and boot ARC (Attached Resource Computer) path. A Windows XP Embedded image can be as small as 5MB. This chapter describes the Windows XP Embedded Architecture and how to create a Windows XP Embedded Image.

B. THE EMBEDDED ARCHITECTURE

Windows XP Embedded components are stored in a SQL Server database called the *Component Database*. The *Component Management Interface* is a COM (Component Object Model [COM]) server that the XP Embedded Studio tools use to access the database. Since the Component Management interface is a COM server, the Windows XP Embedded architecture is object oriented. All "configurations, components, instances, resources, files, registry entries, and repositories" in the architecture are treated as objects. Each component object encapsulates some functionality. A component can be defined as a prototype component, allowing other components to inherit its functionality.

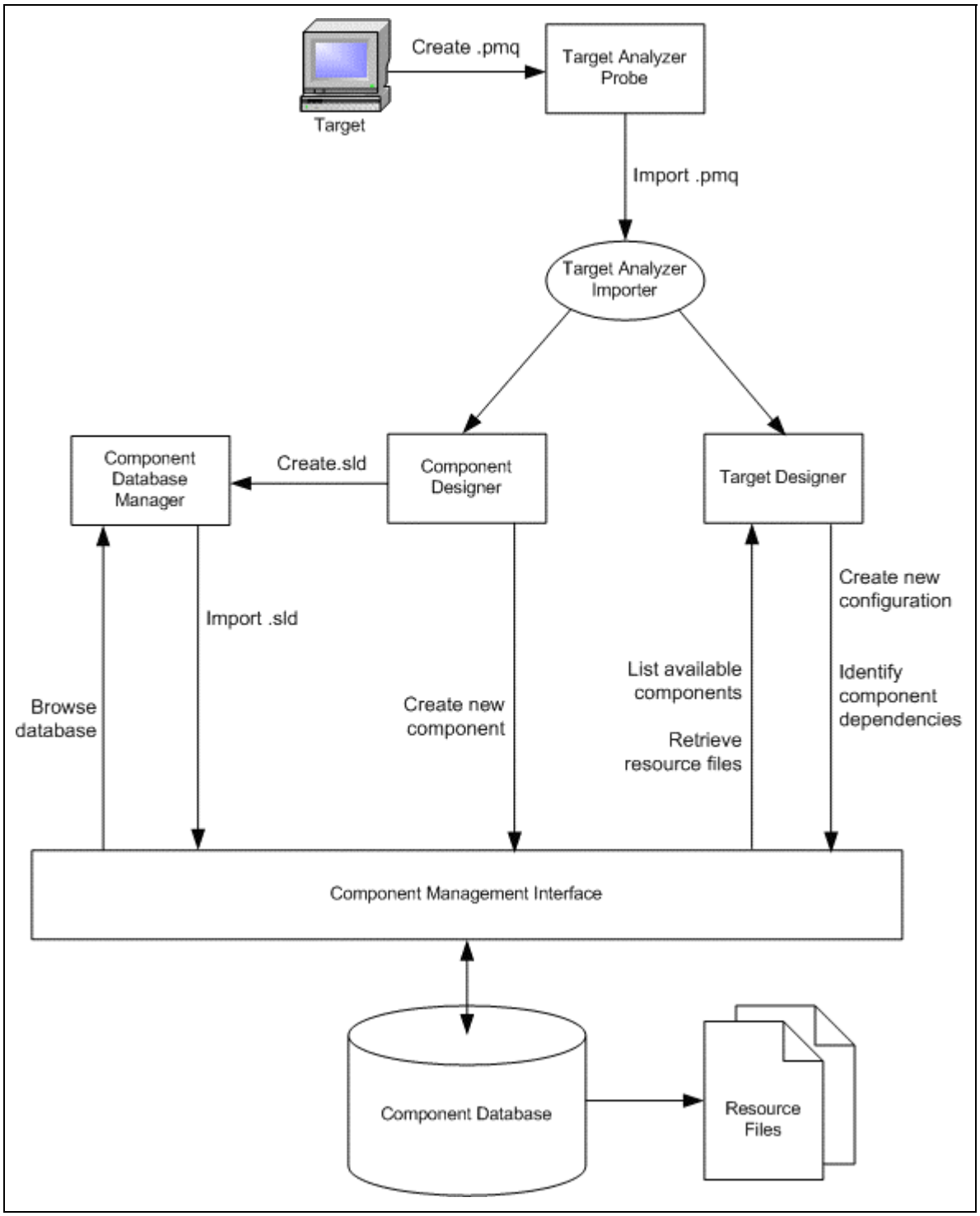

Figure 18. The XP Embedded Architecture

Polymorphism in XP Embedded components "…is usually handled with DHTML [Dynamic Hypertext Markup Language] configuration and build script [FIN1]." This

section describes the architecture of Windows XP Embedded, the tools in the Embedded Development Studio and how a run time image is created (see Figure 18 for a graphical representation.) There are four tools in the Embedded Development Studio [XPE3] that are used to create XP Embedded images: the Target Analyzer [XPE2], the Component Designer [FIN3], the Component Database Manager [FIN4] and the Target Designer [FIN5].

1. Target Analyzer

The Target Analyzer consists of the Target Analyzer Probe and the Target Analyzer Importer. The Target Analyzer Probe (tap.exe or ta.exe) is executed on the target machine and produces a listing of all hardware devices on the target in the form of an XML based .pmq file (pronounced "pumpkin" [FIN6]). The Target Analyzer Importer is a module of the Component Designer and Target Designer. It imports the .pmq file into the Designer application to ensure that the component or configuration includes all of the resources necessary to support the devices of the target.

2. Component Designer

The Component Designer (Figure 19) [FIN3] is used to create custom components, such as a component based on the .pmq file obtained by running the Target Analyzer on a particular target. These components are stored in Source Level Definition (.SLD) XML files that specify the resource files and properties of the component. In addition to the prototype component type used in inheritance relationships, components can also be defined as *macro*, *end-of-life*, *opaque*, *editable*, and/or *allow multiple* type components. A *macro* component defines dependencies on a collection of components but has no resources of its own and is commonly used as a design template. An *end-of-life* component is a component that has become obsolete and should not be included in new configurations but must be retained in the database to support existing configurations. *End-of-life* components can specify which component, if any, replaces it. An *opaque* component does not display its resources in the Target Designer; an editable component allows the developer to edit the component settings in the Target Designer. By default, only one instance of each component can exist in a configuration. An *allow multiple* component will allow multiple instances of a component in one configuration.

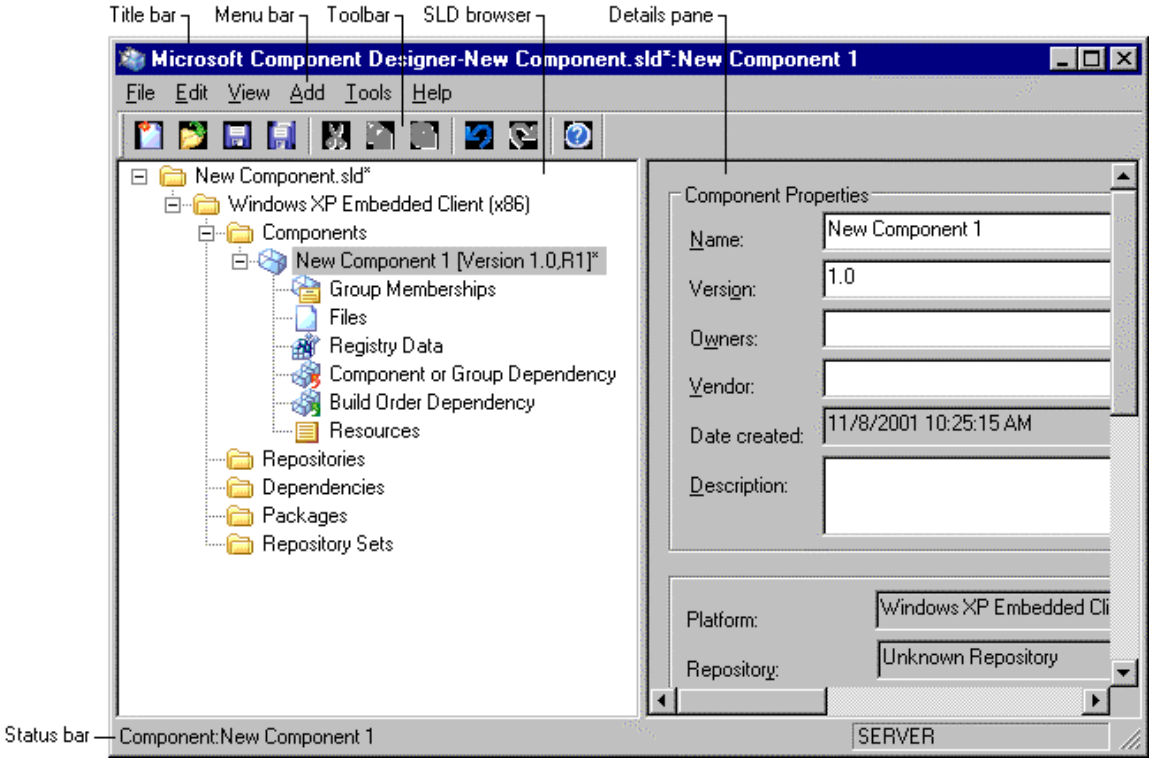

Figure 19. The Component Designer From [XPE2]

3. Component Database Manager

The Component Database Manager [FIN4] uses the Component Management Interface to add new components to the database and display the current components in the database. Once a component has been created, its .sld file must be imported to the component database using the Component Database Manager. Entities may be deleted and the resource file repositories managed from the Component Database Manager.

4. Target Designer

The Target Designer (Figure 20) [FIN5] is used to create configurations, which are stored in .slx files, and build XP Embedded images from those configurations. Components that are available in the Component Database are displayed in the component browser and are added to the configuration by dragging them into the configuration editor pane. There are several default design templates (macro components) available to form the base of XP Embedded configurations: Windows-based Terminal Professional, Information Appliance, Basic Set Top Box, Digital Set Top Box, Advanced

Set Top Box, Kiosk/Gaming Console, Home Gateway, Retail Point of Sale Terminal, and Network Attached Storage.

Each component has a visibility level property, which is a value from 100 to 10,000. The Target Designer also has a visibility level and will only display a component in the browser if the component's visibility level is higher than the Target Designer's visibility level. The default visibility level for components is 1000, macro components are usually set to 2000, and "hidden" components usually have a visibility level of 500. To view all available components in the Target Designer, set its visibility level to 100 [XPE2].

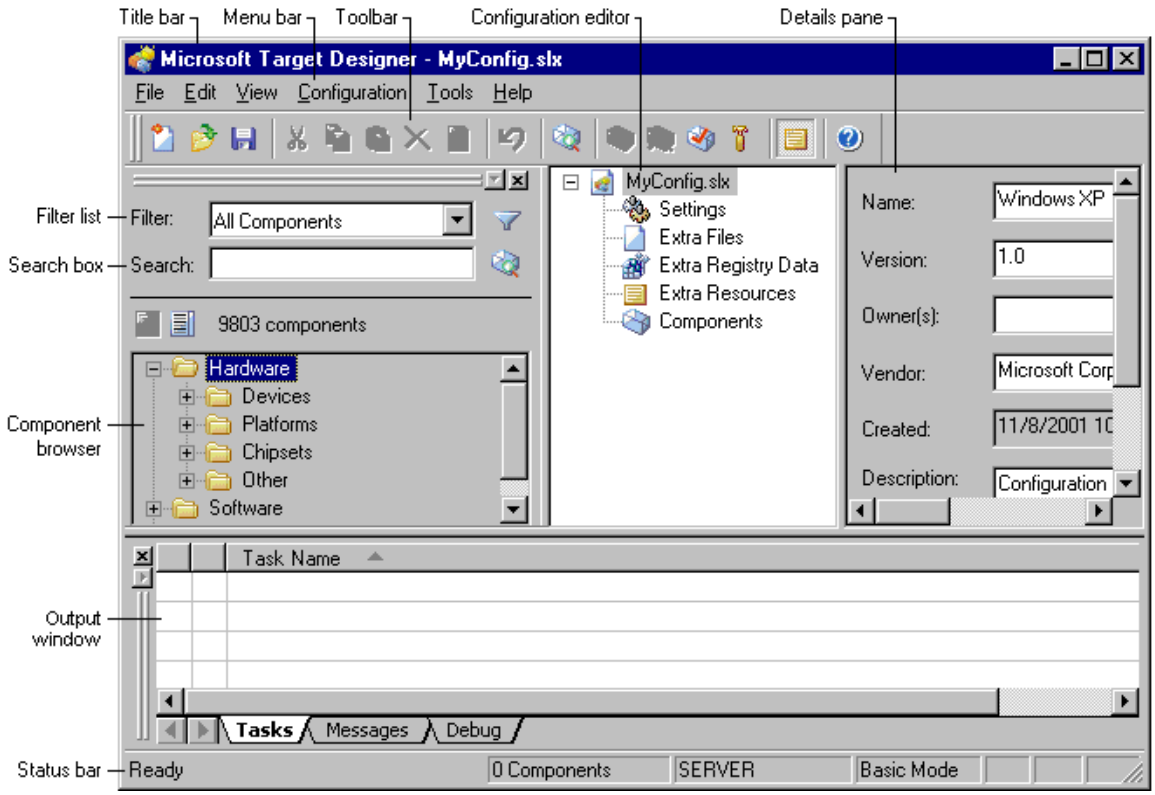

Figure 20. The Target Designer From [XPE2]

5. Embedded Enabling Features

Windows XP Embedded provides several features to enable the building and deployment of XP Embedded images: the First Boot Agent (FBA), Enhanced Write Filter (EWF), System Deployment Image (SDI), El Torito CD Image Preparation Tool, CompactPCI, Message Box Interception, several storage options, Page File Disable, BootPrep, and Power Management Application [XPE2]. The enabling features used in this thesis are the First Boot Agent, Enhanced Write Filter, and the El Torito CD Image Preparation Tool. The FBA runs the first time an XP Embedded image boots and performs configuration tasks that cannot be done offline. The EWF allows an image of XP Embedded to boot from read-only media by storing any write to the media in a memory overlay, which gives the illusion that the media is writable. The El Torito CD Image Preparation Tool converts an XP Embedded run-time image to an ISO 9660 file used in the creation of a bootable CD-ROM image (see Section C.2.) Bootprep (bootprep.exe) modifies the Master Boot Record to point to NTLDR, which is required for booting into Windows XP.

C. BUILDING AN XP EMBEDDED IMAGE

There are several deployment options available to developers who wish to create Windows XP Embedded images. This section describes how to build images with the development environment that was used to demonstrate this thesis. The development environment consisted of a system with a 650 MB FAT partition as the XP Embedded target partition and a partition running Windows XP Professional (Windows 2000 may also be used.) The Windows Embedded Studio was installed on the XP Professional partition. To deploy an XP Embedded image it is handy to create a Windows 98 boot floppy and copy the contents of the Windows Embedded Utilities folder (C:\Program Files\Windows Embedded\Utilities) to another floppy disk. To create a bootable CD image, a CD-RW drive and an application that will burn ISO-9660 (the International Standards Organization standard for CD file systems) images to CDs (Roxio Easy CD Creator was used here) are required. When the Embedded Studio is installed two directories are created: an Embedded Data directory and an Embedded Images directory. The resource files available to components reside in a shared Repository located in C:\Windows Embedded Data. The subverted version of tcpip.sys produced by [LACK03]

was inserted into this repository to create a subverted Windows XP image. In addition to the Repository, a default image directory is created during installation of the Windows Embedded Studio (C:\Windows Embedded Images).

The Windows XP Embedded home page [XPE1] and the Windows XP Embedded Product Documentation [XPE2] were extremely helpful in successfully creating and deploying a Windows XP Embedded image.

1. Creating an XP Embedded Image on a Hard Disk Partition

a. Prepare the Target Partition

Create a 650 MB or less active FAT partition. Boot from the Windows 98 boot floppy and run the Bootprep utility from the utility floppy. Restart into the Windows XP Professional or Windows 2000 partition and run the Target Analyzer (Tap.exe) on the target partition. The Target Analyzer creates a file called DEVICES.pmq containing a listing of the hardware devices on the target.

b. Create a Customized Component for the Target Device

Open the Component Designer and import the DEVICES.pmq file created by the Target Analyzer; this may take a few minutes. Find Devices in the SLD browser (Figure 15) on the left hand side of the Component Designer window and modify the Component Properties. For this demonstration, name the component 'Target Device.' Make the component a 'Selector Prototype Component' and save the new component definition, this will be an .sld file.

c. Import the New Component into the Component Database

Start the Component Database Manager, select Import and choose the .sld file created with the Component Designer in step b. The customized component will now be available for inclusion in a configuration.

d. Create a New Configuration in Target Designer

Open the Target Designer and start a new configuration. The component browser lists all the available components. If a specific component does not appear in the component browser, try lowering the visibility level. Drag the 'Target Device' component to the Configuration Editor (Figure 16) to add it to the configuration. Add the 'Windows-based Terminal Professional' component from the Design Templates to the

configuration. For this demonstration we are utilizing IPSec; click on the filter icon and create a new filter to search for the string 'IP Security' and apply the filter. Add the IP Security Services component and the 'IP Security Tools and User Interface Component.' See Table 2 for a listing of all components used in the demonstration configuration.

Component Name
Target Device
Windows-based Terminal Professional
IP Security Services
IP Security Tools and User Interface
TCP/IP Utilities
Other TCP/IP Utilities
Network Diagnostics
FAT format
FAT
NT Loader

Table 2. XP Embedded Components

Modify the settings of the configuration in the details window to point to the appropriate boot drive and boot ARC path. Set the boot partition size to 650 MB (or the size of the target partition.) Save the configuration, this creates an .slx file. Choose *resolve dependencies*: this iterates through the chosen components and will automatically resolve as many dependencies as possible by querying the Component Management Interface for the dependencies listed in Component Database for each component in the configuration. Any dependencies that cannot be automatically resolved will appear in the Tasks tab of the Output Window (Figure 15). Click on each task and choose a component to resolve the dependency. Choose 'Explorer Shell,' 'NT Loader,' and the appropriate language support component when resolving dependencies.

Find the 'User Interface Core' component in the configuration, this was automatically added during the dependency check. Check the boxes to 'Show Control

Panel on Start Menu' and 'Show My Computer on Start Menu.' Make sure that 'Prohibit Access to Start Menu' and 'Prohibit Access to Hot Keys' is not selected.

e. Build the XP Embedded Image

Select 'Build' and specify an image destination folder within C:\Windows Embedded Images. Run the dependency check again and resolve any new dependencies. Once the build is complete, save the configuration again and exit the Target Designer. If you are running Windows 2000, copy the NTLDR and NTDETECT.com files from your image into the C:\ directory [XPE1].

f. Boot to the XP Embedded Image

To deploy the XP Embedded image, copy the files from the image folder to your target partition. Add a line to the boot.ini file (Start\Run\notepad boot.ini) for the second partition. Restart and choose your XP Embedded partition from the boot menu. The XP Embedded splash screen will appear and the First Boot Agent (FBA) will run; this should take a few minutes. After the FBA restarts the system, select the XP Embedded image again from the boot menu. If the FBA restarts the system again, check the FBALOG.txt file in the image's FBA directory for error messages.

2. Creating an XP Embedded Image on a Bootable CD

Windows XP Embedded includes the El Torito CD Image Preparation Tool to create a bootable CD image of an XP Embedded hard disk image. To create an El Torito image you will need two blank writeable CDs.

a. Create an Enable Auto Layout Registry Component

Create a new component in the Component Designer and name is something like 'EnableAutoLayout.' Right click on 'Registry Data' and choose Add\Registry Data. Enter the information in Table 3 into the 'Add Component Registry Resources' dialog.

Select the 'Default' radio button, save the new .sld file and import the new component to the Component Database using the Component Database Manager.

Root:	HKEY_LOCAL_MACHINE
Key Name:	SOFTWARE\Microsoft\Windows\CurrentVersion\OptimalLayout
Value Name:	EnableAutoLayout
Type:	REG_DWORD
Value:	0

Table 3. Add Component Registry Data

b. Creating an El Torito Configuration

Open the Target Designer and create a new configuration with the following components:

Component Name
Target Device
Windows-based Terminal Professional
IP Security Services
IP Security Tools and User Interface
TCP/IP Utilities
Other TCP/IP Utilities
Network Diagnostics
FAT format
FAT
El Torito CD
EnableAutoLayout
Enhanced Write Filter
EWF Manager Console Application
EWF NTLDR

Table 4. El Torito Configuration Components

Clear the 'Start EWF Enabled' box in the settings of the 'Enhanced Write Filter' component. The Disk Number should be 0 and the Partition Number should be 1. Set the disk image number to 12345678 in the 'El Torito CD' component settings. Modify the 'User Interface Core' component as before. Resolve any dependencies and build the image.

c. Creating the Bootable CD

Copy the image files to the target partition and modify the boot.ini file to include the target partition. Run the HD2ISO utility (hd2iso.exe) to create an ISO image of the target hard disk partition. Select 'Create an ISO-9660/ELTORITO bootable image file' from the HD2ISO main menu. Set the physical drive to the target hard disk and the partition to the target partition. Set the image file path where the ISO-9660 image will be created. In the Advanced Options menu, select 80-minute CD as the target media size, change the signature to 12345678 (the number entered in the El Torito CD component settings), and select the bootable partition. Exit the advanced options menu and select 'Create Image' from the ISO Image menu. Burn the .iso file onto a CD.

d. Configure the Enhanced Write Filter

Insert the CD created in step c. and boot to the XP Embedded hard disk image. The First Boot Agent (FBA) for El Torito images refers to the image on the CD and will reboot the system twice. Once the FBA has finished, boot into the XP Embedded hard disk image and run the EWF (Enhanced Write Filter) manager (using commands ewfmgr and ewfmgr c:). If the EWF manager produces errors, check the FBA log file (FBALOG.txt) for EWF entries.

e. Create an El Torito CD

From the XP Embedded partition on the hard disk run Etprep from a command prompt (etprep /all). This cleans up after the Enhanced Write Filter and swaps the drive letters so that the image runs off of the CDROM. Etprep should reboot the system once. Now create another ISO image of the hard disk partition and burn the image to a CD.

f. Boot From the El Torito CD

Boot any system from the second CD and verify that the EWF is running correctly (by entering ewfmgr C:).

D. SUMMARY

This chapter presented the Windows XP Embedded Architecture and Embedded Development Studio tools. The environment and process used to create Windows XP Embedded Images used in the demonstration of the subversion was described. Chapter V will provide the design of the IPSec attack.

V. AN IPSEC ATTACK

A. INTRODUCTION

This thesis, in cooperation with [LACK03] and [ROG03], demonstrates a subversion of the Windows XP Embedded operating system modeled after the *2-card loader* concept (described in Chapter I.). The subversion is divided into three parts: the artifice base described in [LACK03], the link/loader described in [ROG03] and the attack described in this thesis. The artifice base is the only part of the subversion artifice that is resident in the system. It can be inserted at any phase in the system's lifecycle. The link/loader and the attack are loaded onto a fielded system at a later time. The artifice base allocates memory for the link/loader and provides communication facilities that are used by both the link/loader and the attack. Once the link/loader is running, it loads the attack code into an area of memory that it has allocated separately from the artifice base's memory buffer.

The artifice base and link/loader set the stage for an attack on any kernel module, such as the CryptoAPI. IPSec makes an attractive target since information that users are trying to protect is probably worth stealing. In addition, many IPSec implementations are employed to protect a wide range of application data. This chapter presents the design and implementation of an attack that bypasses IPSec protection but does not exploit an existing flaw in the Windows IPSec implementation.

The development of the IPSec attack assumes that the target device is running a version of Windows XP Embedded subverted by the artifice base component and that the load address of the Windows XP kernel is known. It is emphasized that assumptions of this sort are only for the purpose of simplifying the research setup for this thesis. In a more realistic attack by a professional, the artifice base can first be used to support any needed "casing" of the target to obtain this sort of information before launching a specific attack. The artifice base and link/loader components must run successfully on the target before the attack is loaded. It is also assumed that the target device has sufficient excess memory to load the attack and that the system includes the IP Security Services Component in its configuration. This attack was developed with access to the IPSec

binaries and source using readily accessible software tools including: a kernel debugger, a text editor, an assembler and a disassembler (see Appendix A for a description of the development environment.) Obfuscation of the artifice was not a major design concern since the main purpose of this work is to produce an understandable demonstration of subversion.

B. HIGH LEVEL DESIGN

The attack portion of the artifice consists of attack functions that are loaded on the target machine by the link/loader and a set of scripts that control the attack functions. The attack patches the function of the IPSec driver that handles the sending of IPSec packets. There are several other options for subverting IPSec, such as attacking the keys or the cryptography mechanism. Once the attack is activated, the patch makes a copy of every packet that is to be sent by the target with IPSec protection and sends it out in the clear to be intercepted by the attacker. In practice the attacker would most likely encrypt this data with her own cipher since plaintext may arouse suspicion, but this step was excluded to simplify the operation of the demonstration.

The attack works in two phases. In the first phase, the attack queries the kernel for the loaded module list to find the load address of the IPSec driver. The load address of the kernel must be known, this could be returned as a function of the artifice base. In the second phase, the address of the target function is calculated from the load address of the driver and passed to the loaded attack through a trigger. The target function is then patched with the attack code. Once the attack is activated, all data to be sent over IPSec will also be sent in the clear and intercepted by the attacker.

C. INTERFACE DESCRIPTION

The attack component must provide a user interface as well as a communication interface with the link/loader component from [ROG03] and the artifice base from [LACK03].

1. User Interface

To run the IPSec attack on a target machine the attacker runs a set of scripts on a remote computer that control the artifice. The scripts load the second phase of the artifice base, load the attack, set attack triggers, and run each trigger. The artifice base provides

the triggering mechanism. A trigger is set associating a function number with the offset of a call instruction to an attack function, since the artifice base only sets triggers to a 4 byte offset. When a trigger packet is received denoting a function number, the artifice base will call the offset associated with that function number, which will in turn call the attack function. For this attack, triggers will be set for the *findModules* function, the *patching* function, and the *activate/deactivate* function. The packets containing data exfiltrated by the attack will be captured and displayed by a packet sniffer such as Ethereal [ETHER].

2. Link/Loader Interface

The binary files of the attack must be sent to the link/loader. The link/loader relocates the attack functions to the target machine. Since the link/loader may not be able to allocate sufficient contiguous memory, the attack functions refer to a jump table that contains the absolute addresses of each attack function. The link/loader completes this table when the load addresses of the functions are known. The completed jump table is also used to set triggers for the loaded attack functions.

3. Artifice Base Interface

The artifice base subverts the TCP/IP driver such that the artifice base intercepts malformed UDP packets with a specific bad checksum. The user scripts use a tool such as *Sendip* from Project Purple [PURPLE] to create these packets. The attack functions communicate with the artifice base through an internal call to the artifice base's *feedback* function that is used to send the exfiltrated information as data in ICMP packets.

D. DETAILED DESIGN

1. The Attack

The IPSec attack patches itself into a function that executes every time a packet is sent with IPSec protection. Once the target function is patched, it will jump to the attack patch code that sends a copy of the packet using the *feedback* function provided by the bootstrap.

a. Data

The Attack code was written in assembly with a small memory model. The attack functions reference a global data table (Table 5) that contains the jump table

and a data table. The jump table contains absolute addresses of the functions provided by the attack and is completed by the link/loader once the load address of each function on the target device is known. The global data table is used to provide position independence to the attack functions. A reference to the artifice base must be stored to allow the attack to call the *feedback* function and access the data in trigger packets. The attack has a global status variable that indicates whether the current IPSec packet should be copied and sent (activated state) or whether the attack should do nothing (deactivated state). The address of the target function is also saved to the global data table for use by all attack functions.

Jump Table	Find Modules Function Address (requires trigger)
	Activate/Deactivate Function Address (requires trigger)
	Patching Function Address (requires trigger)
	Patch Function Address
Data Table	Artifice Base Feedback Reference
	Status Variable
	Target Function Address
	Artifice Base Trigger Reference

Table 5. The Global Data Table

b. Functions

The IPSec attack implements four main functions: a *findModules* function, the *patching* function, an *activate*/deactivate function and the actual IPSec attack *patch*. The *findModules* function (see Table 6) queries the kernel for the loaded module list and sends each entry out with the artifice base's *feedback* function. The *patching* function (see Table 7) swaps the first few bytes of the target function with an instruction to jump to the attack *patch* and inserts an instruction to jump back to the patched IPSec function at the end of the attack patch. The *activate*/*deactivate* function (see Table 8) sets the value of the global activation status variable, which the attack patch verifies before performing any action. If the status is active (the variable is set to "1"), the exploit patch

(see Table 9) copies the packet to the artifice base's *feedback* buffer. Once the packet is copied, the patch calls the artifice base's *feedback* function to exfiltrate the packet.

The *findModules* function (Table 6) calls the *ExAllocatePool* and *NtQuerySystemInformation* functions exported by the kernel (ntoskrnl.exe). The virtual address of the kernel must be known in order to determine the address of these two functions. *ExAllocatePool* allocates a buffer that is passed to *NtQuerySystemInformation* along with the type of information requested. In this case we are interested in the loaded module list. Once the module list is returned in the buffer, the *findModules* function sends each entry of the loaded module list out with the artifice base's *feedback* function.

Function Name:	FindModules
Description:	Calls the functions ExAllocatePool and NtQuerySystemInformation to exfiltrated the loaded module list
Executed by:	Trigger
Preconditions:	The load address of ntoskrnl.exe is known
Postconditions:	The load address of ipsec.sys (and all other kernel modules) is known

Table 6. Find Modules Function

The target IPSec function is at a set offset from the start of the IPSec driver, which is returned by the *findModules* function. This address is computed and sent in the trigger to the *patching* function, which saves it to the global data table (Table 5.) The patching function (Table 7) stores the first 9 bytes (three instructions) of the target function into the first 9 bytes of the patch function and sets the first instruction of the target function to jump to the address of the patch function. The address of the patch function is retrieved from the global data table. The last 7 bytes of the patch function are an instruction to jump back to the instruction in the target function following the three that were displaced.

Function Name:	Patching
Description:	Patches the target function with the attack patch
Executed by:	Trigger
Parameters:	The address of the target IPSec function (passed in the trigger)
Preconditions:	The find modules function has executed and the address of the target function has been calculated
Postconditions:	The target function is patched

Table 7. Patching Function

The activate/deactivate function (Table 8) is executed by the bootstrap in response to a trigger that is set by the user scripts. It sets the status variable to the value sent in the data field of the trigger packet (0 for deactivate, 1 for activate). See Section D.2 of this chapter for a description of the packet fields.

Function Name:	Activate/Deactivate
Description:	Sets the value of the status variable
Executed by:	Trigger
Parameters:	Status value (passed in trigger)
Preconditions:	Attack has been patched into target, status is deactivated by default
Postconditions:	The status variable is set to the value in the data field of the trigger packet

Table 8. Activate/Deactivate Function

The patch function (Table 9) performs the actual attack. The pointer to the packet is retrieved from the stack and the clear text data of the packet is copied to the artifice base's *feedback* buffer. Once the data has been copied, the *feedback* function is called to exfiltrate the data. The patch jumps back to the target function and IPSec communication on the subverted machine continues without interruption.

Function Name:	Patch
Description:	Copies the contents of the IP packet buffer into the artifice base's output buffer, then calls the artifice base's *feedback* function to exfiltrate the packet
Executed by:	Target Function
Preconditions:	Status is active and the addresses of the IP packet buffer and the artifice base's output buffer are stored in the global data table
Postconditions:	The clear text data is exfiltrated and the IPSec communication uninterrupted

Table 9. IPSec Patch Function

c. Attack Package

The link/loader needs to receive the attack and information required to link the attack in a recognized format. This attack package contains: the name of the executable attack file, the length of the code, and the length of the jump table.

2. The User Scripts

The user scripts compose packets with a specific bad UDP checksum that are sent to the target machine and interpreted by the artifice base. There are two types of packets recognized by the artifice base: a run function packet and a set trigger packet. A run function packet (Figure 21) will cause the artifice base to call the address associated with the function number supplied in the function number field of the artifice base header. A set trigger packet (Figure 22) sets the association of the number supplied in the trigger number field with the offset supplied in the Offset/Jump field.

The artifice base reserves the first three function numbers. Function number 0 does nothing. Function number 1 denotes the *load* instruction, which copies the contents of the data field (up to the specified length) into the artifice base's memory buffer at the offset specified. Function number 2 specifies the *set trigger* function, which associates the trigger number with the offset in the jump field. For the IPSec attack, triggers will be set to run the *findModules* function, the *patching* function, and the *activate*/deactivate function.

<table>
<tr><td colspan="6">32 bits</td></tr>
<tr><td colspan="6">UDP Header</td></tr>
<tr><td colspan="4">Source</td><td colspan="2">Destination</td></tr>
<tr><td colspan="4">Length</td><td colspan="2">Bad Checksum</td></tr>
<tr><td colspan="6">Bootstrap Header</td></tr>
<tr><td colspan="6">Session ID
(32 bits)</td></tr>
<tr><td>Feedback
(1 bit)</td><td>Unused
(3 bits)</td><td>Function Number
(4 bits)</td><td>Unused
(8 bits)</td><td colspan="2">Offset
(16 bits)</td></tr>
<tr><td colspan="4">Length
(16 bits)</td><td colspan="2">Data
(variable length)</td></tr>
</table>

Figure 21. A Run Trigger Packet

<table>
<tr><td colspan="6">32 bits</td></tr>
<tr><td colspan="6">UDP Header</td></tr>
<tr><td colspan="4">Source</td><td colspan="2">Destination</td></tr>
<tr><td colspan="4">Length</td><td colspan="2">Bad Checksum</td></tr>
<tr><td colspan="6">Bootstrap Header</td></tr>
<tr><td colspan="6">Session ID
(32 bits)</td></tr>
<tr><td>Feedback
(1 bit)</td><td>Unused
(3 bits)</td><td>Function Number
(4 bits)</td><td>Trigger Number
(8 bits)</td><td colspan="2">Jump
(16 bits)</td></tr>
<tr><td colspan="4">Length
(16 bits)</td><td colspan="2">Checksum
(16 bits)</td></tr>
<tr><td colspan="6">Data
(variable length)</td></tr>
</table>

Figure 22. A Set Trigger Packet

3. Viewing the Exfiltrated Data

To view the data exfiltrated by the attack patch, the attacker uses a packet sniffer such as Ethereal [ETHER]. Ethereal allows users to filter by IP address and protocol. The artifice base's *feedback* function sends ICMP packets, which can easily be located with the filters and sorting functions of Ethereal.

C. FUTURE WORK

This attack establishes a framework that can be used to attack other kernel modules in a subverted operating system. Future work on this attack could include the addition of an un-patching function that, when triggered, would reset the target function so that it would no longer jump to the attack patch. Some environments (such as military applications) may be protected by only allowing inbound communication. The attack could be designed to operate without feedback capabilities in this case. The *findModules* function could be modified to only send information on a specific module or to set the target function address in the global data table without using the *feedback* function. The *findModules* function could also be modified to exfiltrate other system information provided by *NtQuerySystemInformation*. Instead of requiring the artifice base to provide the load address of the kernel, the attack could make an educated guess and search memory for the necessary kernel functions using a pattern matching technique. The attack could also be written to search the kernel's internal data structures directly, such as the loaded module list used by *NtQuerySystemInformation* or the kernel handle table [Page 141 of SOL00], to find the load address of the target module. The link/loader could be extended to provide persistence of the attack *patch* between system boots.

D. SUMMARY

The artifice base, link/loader and attack components of the subversion demonstrate the ability for a mere six lines of code [LACK03] to dynamically load in additional malicious code. This is a flexible attack that does not require the attack writer to know where the attack code will be loaded in memory on the target device. The attack writer also does not need to know what applications will be running on the target machine when the artifice base is inserted. The attack demonstrates that the security of applications is inconsequential and ineffective if the underlying operating system has

been subverted. Since the IPSec encryption mechanism is bypassed, the strength of the encryption algorithm is also inconsequential. This attack does not exploit an existing flaw in IPSec. Instead, it uses the privileges given to it by the artifice base to modify the application to meet the attacker's objectives.

VI. CONCLUSION

An attack on the Windows XP Embedded operating system implementation of IPSec was demonstrated using a dynamic subversion artifice modeled after the *2-card loader* concept [SCH03]. The attack can be loaded onto a fielded system that has been subverted by the six lines of code comprising the *artifice base* [LACK03], which could be inserted into the kernel at any phase in the system's lifecycle. The attack provides a flexible method for the attacker, who may not be the same individual who inserted the artifice, to gain total control of the subverted system. Due to the dynamic loading property of this subversion, the attacker does not have to decide the aspect of the system to be targeted until a time of her choice. Although IPSec was chosen for this demonstration, a strategy was presented for the subversion of any kernel module.

The attack does not exploit an existing flaw in the target module but is possible because the *artifice base* is inserted into the kernel of an operating system for which adversaries have access to source code directly or indirectly, e.g., via reverse engineering of a normal commercial product distribution. No amount of additive security measures would hinder the operation of the subversion artifice since its location in the most privileged portion of the system allows all security mechanisms to be bypassed.

Known methods for developing systems with verified protection so they can be independently evaluated to establish that they are free from subversion were discussed and several projects that utilized these methods were presented. However, nearly all commercial operating systems are currently not developed with these methods, so there is no guarantee that they do not currently contain subversive artifacts. In fact, there is evidence [THU01] that terrorist groups have recognized the power of subversion attacks and have at least attempted their implementation.

PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: IMPLEMENTING THE ATTACK

A. INTRODUCTION

This appendix describes the development and test environments for the IPSec attack as well as the implementation of the subversion demonstration. The attack component of the exploit was written on a Windows XP platform in assembly language using TextPad [TEXT] and assembled with MASM (Microsoft's Macro ASseMbler) [IRVI03]. PEBrowse Interactive [SMIDGE] was used to debug and prepare the attack scripts, which use SendIP [PURPLE] to create and send packets to the target computer. Once the attack is loaded on the target computer, SoftICE [ICE] (a kernel debugger specifically for Microsoft operating systems) is used to step through the execution of the artifice. Ethereal [ETHER] (a packet sniffer) is used to capture the exfiltrated packets sent by the target machine.

B. THE ASSEMBLY ENVIRONMENT

1. Configuring MASM and TextPad

Install MASM [IRVI03] and TextPad [TEXT]. Configure the TextPad tools menu to include the commands for building and running MASM programs. To add the build command to the tools menu, go to *Configure* and select *Preferences*. Click on *Tools* and Add a "DOS command…." Enter 'make32.bat $BaseName' and click OK. Rename the command to 'Build 32-bit MASM' and click Apply. Expand *Tools* in the left-hand pane and enter '$FileDir' as the *Initial folder* (see Figure 23.) To add the run MASM tool, select Tools in the left-hand pane and click Add. Choose "DOS command…," enter '$BaseName' and click OK. Rename the command 'Run ASM Prog.' Expand *Tools*, select *Run ASM Prog* and uncheck *Capture output* (see Figure 24) [IRVI03].

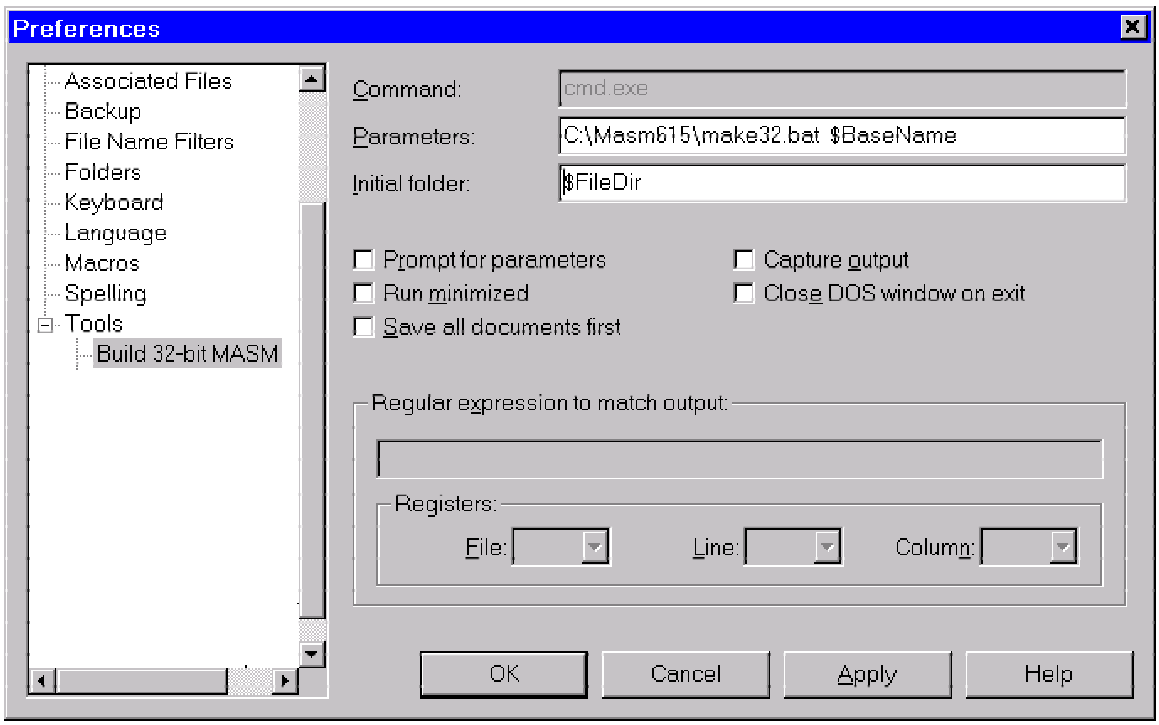

Figure 23. Configure TextPad to Build MASM From [IRVI03]

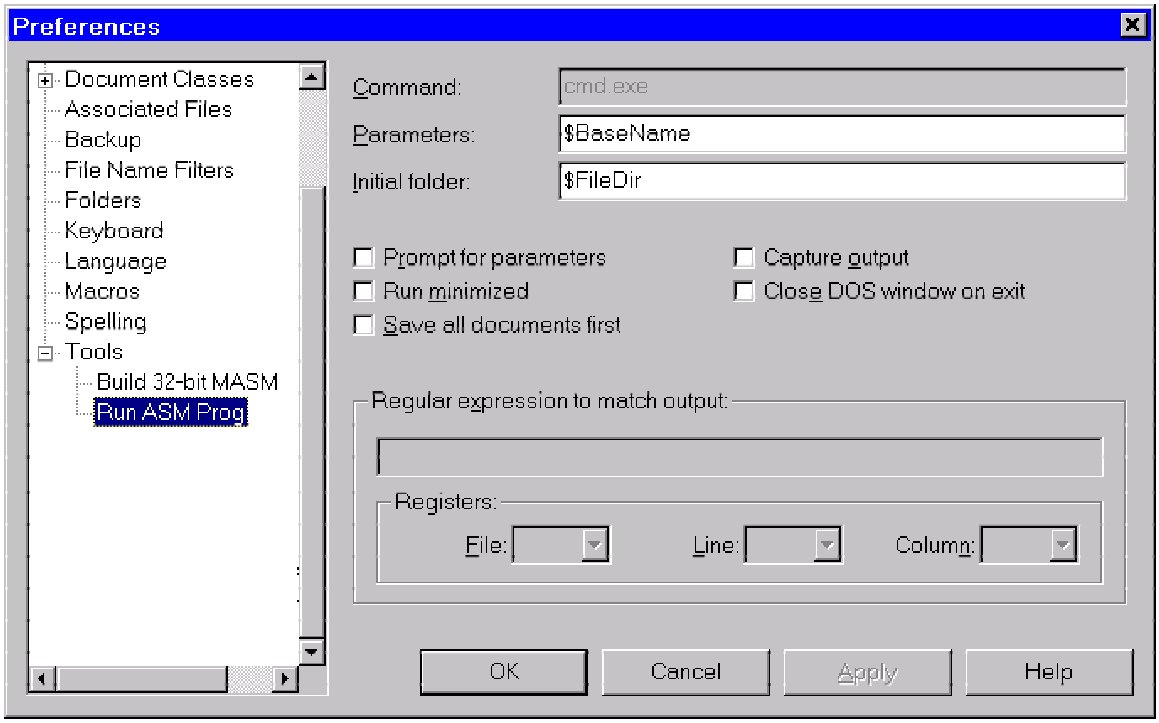

Figure 24. Configure TextPad to Run ASM Program From [IRVI03]

2. Writing the Attack

The link/loader patches the absolute (virtual) address of the global data table into the same location at the beginning of each function. The patch function needs to leave 9 bytes at the start of the function to execute the displaced instructions from the target function. The instruction that loads the address of the global data table in all functions need to be aligned through the insertion of 9 nop instructions (see Figure 25.)

To jump to an address stored in the jump table, load the effective address of the desired entry in the jump table into a register. Then move the contents of that address into a register and jump to the register (see Figure 25.)

To access local variables in position independent code, call the label of the next instruction and then pop the address from the stack to find the current absolute address. Add or subtract the offset of the local variables from this address (see Figure 25.)

3. Debugging with PEBrowse

Install PE Browse Interactive (www.smidgeonsoft.com.) Choose *Start Debugging* from the File menu and select the executable file to debug. Use the index pane on the left-hand side to browse the sections of the PE (Portable Executable) file. Right click on the .text section and choose *Disassemble* to see the disassembly, or *Dump* to view the hex dump. Step through the execution with the F10 function key. To view the disassembly of a kernel module, choose *Load Module* from the File menu. The sections of the PE file will appear in the index pane as before.

4. Preparing the Scripts

To get the opcodes of the attack function into a script, copy the dump of the .text section of the executable file into a TextPad document. The copy function of PE Browse copies the whole section. Use TextPad's search and replace by regular expression to strip out the line numbers, ASCII, and all spaces. Copy the hex into a script to create a load packet. See Figure 26 for a sample script using SendIP to create a load packet.

Alignment of the global data table address	

`Nop`
`nop`
`...`
`nop`
`nop`
	Nine nop instructions for alignment

`Pushad` `pushfd`	Save all registers and flags
`Mov ebx, 12345678h`	Move the address of global data table into a register
Using the jump table	
`Lea ecx, [ebx + 4]`	Load the effective address of the second entry of the jump table into a register (each entry is 4 bytes long)
`Mov ecx, [ecx]`	Move the contents of the second entry into a register
`Jmp ecx`	Jump to the address in the second entry of the jump table
Accessing local variables	
`Var1 DWORD 0111h` `var2 DWORD 0222h`	Variable declaration
`call L1` `L1: pop ebx`	Call the next instruction (at label L1) Pop the last entry on the stack into a register; this is the location of the label L1. Access the local variables by subtracting their offset from this location.

Figure 25. The Attack Function Template

```
#!/bin/bash
sendip -v -d 0x0403020181000000000CAABBCCDDEEFFAABBCCDDEEFF
-p ipv4 -is <source IP> -id <destination IP>
-p udp -us 500 -ud 53 -uc 58391 <target IP>
```

Field	Value
Session ID	04030201
Feedback (0x1000)	8
Function number (Load)	1
Unused	00
Offset	0000
Length	000C
Data	AABBCCDDEEFFAABBCCDDEEFF

Figure 26. SendIP Script for a Load Packet

C. THE TEST ENVIRONMENT

The test environment consists of the subverted *target* machine that has an IPSec connection established with the *bystander* machine and the *attacker* machine, which communicates with the artifice and intercepts the exfiltrated data.

1. Using SoftICE

The SoftICE kernel debugger [ICE] was used to step through the attack on the subverted *target*. To load the symbols for the subverted tcpip.sys file, run a checked build of the driver and copy all source files and the .pdb file to the target computer. Open SoftICE's Symbol Loader and open tcpip.sys. Click on the Load icon. Set a breakpoint at the start of the artifice base function. See Table 10 for a listing of useful SoftICE commands.

Ctrl-D	View SoftICE
lines	Change the length of the SoftICE window
width	Change the width of the SoftICE window
bpx	Set breakpoint on execution
bl	List breakpoints
bd	Disable breakpoint
bc	Clear breakpoint
wd	Watch data
D	Display
F8	Step into
F10	Step over
G	Go

Table 10. Useful SoftICE Commands

2. Executing the Attack

Open Ethereal and start capturing packets. Load the second phase of the artifice base and all attack functions. Set the triggers and run the findModule function. Stop the Ethereal capture and filter by the IP address of the target computer. Sort by protocol and locate the ICMP packet that contains the module entry for IPSec.sys (see Figure 27.) Calculate the address of the target function from the load address that is returned and enter that address in the data field of the patching function's trigger packet. Run the patching function and activate the patch. Start another capture with Ethereal. Open an FTP connection between the *target* and the *bystander* and transfer a file. Figure 28 shows the Ethereal capture of a file containing the text "This file contains sensitive information" being transferred between the subverted target computer and the bystander over an IPSec connection.

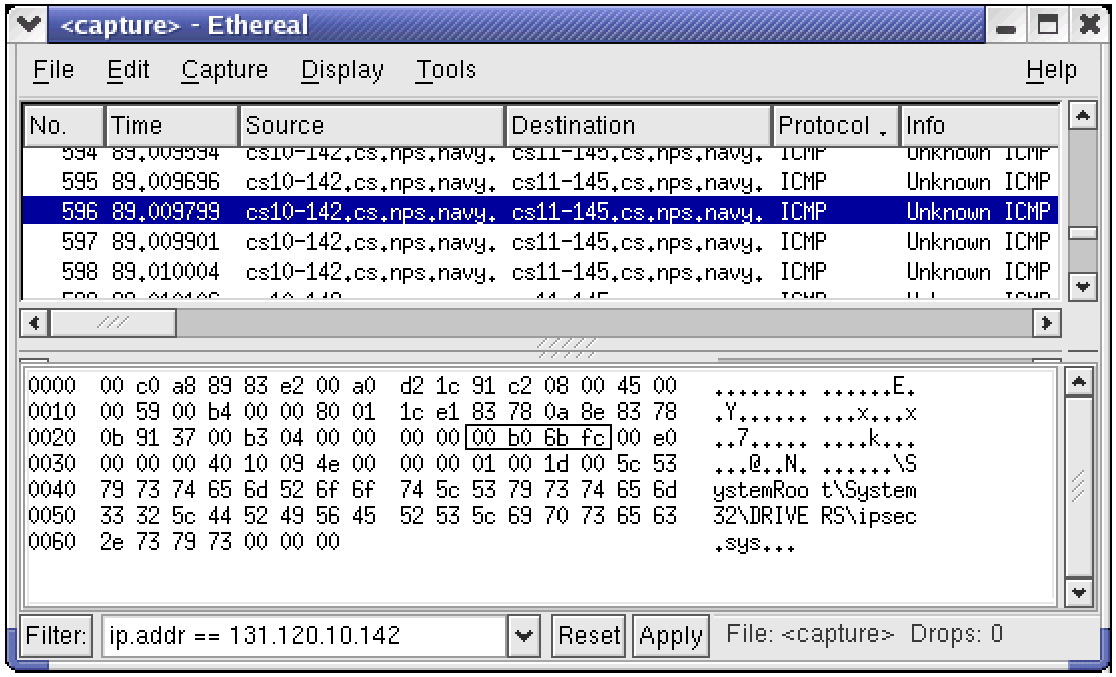

Figure 27. Exfiltrated Module List Entry

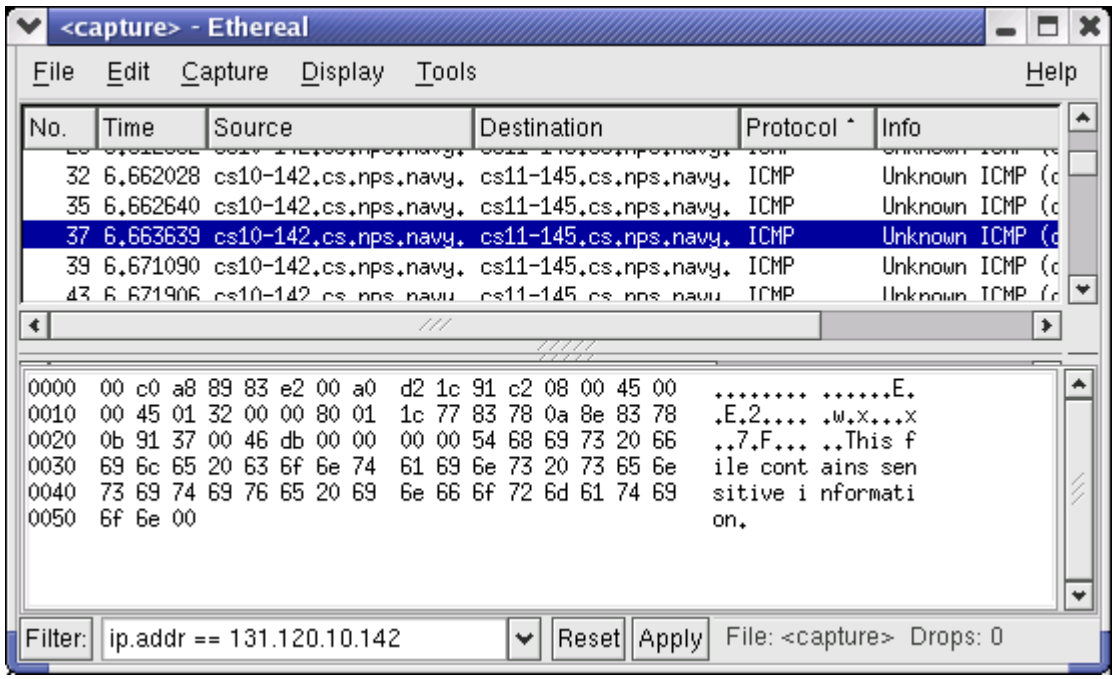

Figure 28. Exfiltrated Clear Text

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[ABR95] Abrams, M., Jajodia, S, and Podell, H., Eds., *Information Security:An Integrated Collection of Essays*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

[AME83] Ames, S.R., Gasser, M., and Schell, R., "Security Kernel Design and Implementation: An Introduction." *IEEE Computer*, Vol. 16, No. 7, pp. 14-22, July 1983.

[AND02] Anderson, E.A., *A Demonstration of the Subversion Threat: Facing a Critical Responsibility in the Defense of Cyberspace*, Master of Science Thesis, Naval Postgraduate School, Monterey, CA, March 2002.

[BEL73] Bell, D., LaPadula, L., "Secure Computer Systems: Mathematical Foundations and Model." *MITRE Report*, MTR 2547, Vol. 2, November 1973.

[BELL90] Bell, D. E., "Lattices, Policies, and Implementations," *Procedures of the 13th National Computer Security Conference,* Washington, D.C., pp. 165-171, October 1-4,1990.

[BELL91] Bell, D. E., "Putting Policy Commonalities to Work." *Procedures of the 14th National Computer Security Conference,* Washington, D.C., pp. 456-471, 1991.

[BLP75] Bell, D. E., and La Padula, L., "Secure Computer System: Unified Exposition and Multics Interpretation," ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA, 1975.

[BIBA77] Biba, K., "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, ESD/AFSC, Hanscom AFB, Bedford, MA, April 1977.

[BRI95] Brinkley, D.L., and Schell, R.R., "What is there to Worry About? (An Introduction to the Computer Security Problem)," in *Information Security: An Integrated Collection of Essays*, IEEE Computer Society Press, Los Alamitos, CA, pp. 11-39, 1995.

[BOE91] National Security Agency, National Computer Security Center, "Final Evaluation Report, Boeing Corporation, MLS LAN," CSC-EPL-91/005. C-Evaluation No. 02-92, August 28, 1991, http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-94-006.html, June 1, 2003.

[BRN89] Brewer, D., and Nash, M., "The Chinese Wall Security Policy," *Procedures of the 1989 IEEE Symposium on Security and Privacy,* Oakland, CA, pp. 206-214, May 1989.

[CC99] The Common Criteria Project, "Common Criteria for Information Technology Security Evaluations," Version 2.1, CCIMB-99-031, August 1999, http://www.commoncriteria.org/cc/cc.html, June 9, 2003.

[CG01] The Cable Guy, "Exploring Peer-to-Peer IPSec in Windows 2000," *Microsoft TechNet,* May 2001, http://www.microsoft.com/technet/columns/cableguy/cg0501.asp, April10, 2003.

[CG02] The Cable Guy, "IKE Negotiation for IPSec Security Associations," *Microsoft TechNet*, June 2002, http://www.microsoft.com/columns/cableguy/cg0501.asp, April 4, 2003.

[CHE81] Cheheyl, M.H., Gasser, M., Huff, G.A., Millen, J.K., "Verifying Security," *ACM Computing Surveys*, Vol. 13, No. 3, pp. 279 – 339, September 1981.

[CLW87] Clark, D.D., and Wilson, D.R., "A Comparison of Commercial and Military Computer Security Policies," *Procedures of the 1987 IEEE Symposium on Security and Privacy,* Oakland, CA, pp. 184-194, April 27-29, 1987.

[COH03] Cohen, F., "Cyber-Risks and Critical Infrastructures," February 27, 2003, http://all.net, March 3, 2003.

[COM] Microsoft Corporation, "The Component Object Model," 2003, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/com_757w.asp, June 9, 2003.

[DEN76] Denning, Dorothy E. "A Lattice Model of Secure Information Flow." *Communications of the ACM*. Vol. 19, No. 5, pp. 236-243, May 1976.

[DIFF76] Diffie, W., and Hellman, M., "New Directions in Cryptography." *IEEE Transactions on Information Theory.* Vol. 22, No. 6, pp. 644-654, November 1976.

[DIJ68] Dijkstra, E.W., "The Structure of the "THE"-Multiprogramming System." *ACM Symposium on Operating Systems Principles*, Vol. 11, No. 5, pp. 341-345, May 1968.

[DOD85] National Security Agency, National Computer Security Center, *Trusted Computer System Evaluation Criteria,* DOD 5200.28-STD, December 1985.

[DOD87] National Security Agency, National Computer Security Center, *Trusted Network Interpretation*, NCSC-TG-005 Version 1, July 31, 1987.

[DOR99] Doraswamy, N., and Harkins, D., *IPSec: The New Security Standard for the Internet, Intranets and Virtual Private Networks*, Prentice Hall. 1999.

[ETHER] "The Ethereal Network Analyzer," May 24, 2003, http://www.ethereal.com, June 9, 2003.

[FIN0] Fincher, J., "Getting to Know Windows NT Embedded and Windows XP Embedded," November 20, 2001, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded11202001.asp. April 28, 2003.

[FIN1] Fincher, J., "Windows XP Embedded Architecture Basics." December 18, 2001, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded12182001.asp, April 28, 2003.

[FIN2] Fincher, J., "Component Designer: Casting the Mold, Part 1," January 15, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded01152002.asp, April 28, 2003./

[FIN3] Fincher, J., "Component Designer: Casting the Mold, Part 2," February 18, 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded02182002.asp, April 28, 2003.

[FIN3] Fincher, J., "Component Designer: Pulling It All Together," June 19, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded06192002.asp, April 28, 2003.

[FIN4] Fincher, J., "Component Database Manager," July 19, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded07192002.asp, April 28, 2003.

[FIN5] Fincher, J., "Target Designer, Inside and Out," August 22, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded08222002.asp, April 28, 2003.

[FIN6] Fincher, J., "Running Free: Runtime Basics," September 26, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded09262002.asp, April 28, 2003.

[FIN7] Fincher, J., "The New Addition: Windows XP Embedded with Service Pack 1," December 19, 2002, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnembedded/html/embedded12172002.asp, April 28, 2003.

[FRA83] Fraim, L.J., "SCOMP: a solution to the MLS problem," IEEE Computer, Vol. 16, No. 7, pp. 26-46, July, 1983.

[GEMSOS] National Security Agency, National Computer Security Center, "Final Evaluation Report, Gemini Trusted Network Processor" CSC-EPL-94/008, September 6, 1994, http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-94-008.html, June 9, 2003.

[HRU76] Harrison, M., Ruzzo, W., and Ullman J., "Protection in Operating Systems." *Communications of the ACM*, Vol. 19, No. 8, pp. 461-471, August 1976.

[ICE] NuMega from Compuware. *SoftICE Driver Suite,* 2003, http://www.compuware.com/products/driverstudio/softice.htm, June 13, 2003.

[IRV01] Irvine, C.E., and Levin, T.E., "Data Integrity Limitations in Highly Secure Systems," *Proceedings of the International System Security Engineering Association Conference*, Orlando, FL, March 1, 2001.

[IRV03] Irvine, C.E., Levin, T.E., and Dinolt, G.W., "Trusted Computing Exemplar Project," Center for Information Systems Security Studies and Research, Naval Postgraduate School, Monterey, CA, 2003, http://cisr.nps.navy.mil/projecttcx.html.

[IRVI03] Irvine, K. R., *Assembly Language For Intel-Based Computers*, Fourth Edition. Prentice Hall. Upper Saddle River, NJ. 2003.

[JAN76] Janson, P.A., "Using Type Extension to Organize Virtual Memory Mechanisms," MIT-LCS-TR-167, Laboratory for Computer Science, M.I.T., Cambridge, MA., September 1976.

[KAR74] Karger, P. A., and Schell, R. R., "Multics Security Evaluation: Vulnerability Analysis," ESD-TR-74-193 Vol. II, ESD/AFSC, Hanscom AFB, Bedford, MA, June 1974.

[KAR02] Karger, P.A., and Schell, R.R., Thirty Years Later: Lessons Learned from the Multics Security Evaluation, IBM Research Report RC22534 (W0207-134) July 2002. *Proceedings of the Annual Computer Security Application Conference*, December 2002.

[KAR91] Karger, P. A., Zurko, M. E., Bonin, D. W., Mason, A. H., and Kahn, C. E., "A Retrospective of the VMM Security Kernel," *IEEE Transactions on Software Engineering,* Vol. 17, No. 11, November 1991.

[KEA00] Keaten, J., "Microsoft: Big Hack Attack," *CNN Money*, October 27, 2000, http://money.cnn.com/2000/10/27/technology/microsoft, March 3, 2003.

[LACK03] Lack, L., *Using the Bootstrap Concept to Build an Adaptable and Compact Subversion Artifice*. Master's Thesis. Naval Postgraduate School. Monterey, California. June 2003.

[LAMP73] Lampson, B. W., "A Note on the Confinement Problem," *Communications of the ACM*, Vol. 16, No. 5, pp. 279-285, October 1973.

[LAN81] Landwehr, C. E., "A Survey of Formal Models for Computer Security," Computer Science and Systems Branch, Information Technology Division, Naval Research Laboratory, Washington, D.C., September 30, 1981.

[LIP82] Lipner, S. "Non-Discretionary Controls for Commercial Applications," *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, Oakland, CA, pp. 2-10, April 1982.

[MIL79] Millen, J.K., Huff, G.A., and Gasser, M. "Flow Table Generator," MITRE Working Paper, WP-22554, The MITRE Corporation, Bedford, MA, November 1979.

[MYE80] Myers, P.A., *Subversion: The Neglected Aspect of Computer Security*, Master of Science Thesis, Naval Postgraduate School, Monterey, CA, June 1980.

[NSPPCI] The Whitehouse, "The National Strategy for the Physical Protection of Critical Infrastructures and Key Assets," February, 2003, www.whitehouse.gov/pcipb/physical.html, March 3, 2003.

[PAR72] Parnas, D.L., "A Technique for Software Module Specification with Examples," *Communications of the ACM*, Vol. 15, No. 5, pp. 330-360, May 1972.

[PAR96] Parnas, D.L., "Why Software Jewels are Rare," *IEEE Computer,* Vol. 29, No. 2, pp. 57-60, February 1996.

[PCCIP97] Report to the President's Commission on Critical Infrastructure Protection. "Threat and Vulnerability Model for Information Security," 1997, www.cert.org/archive/pdf/2-97sr003.pdf, March 3, 2003.

[PERRINE] Perrine, T., Codd, J., and Hardy, B., "An Overview of the Kernelized Secure Operating System (KSOS)," http://users.sdsc.edu/~tep/Presentations/Overview_Paper.text. October 31, 2002.

[PURPLE] Ricketts, M., "SendIP – Programs – Project Purple." April 21, 2003, http://www.earth.li/projectpurple/progs/sendip.html, June 9, 2003.

[REED79] Reed, D.P., Kanodia, R.K., "Synchronization with Eventcounts and Sequencers," *Communications of the ACM*, Vol. 22, No. 2, February 1979.

[ROG03] Rogers, D., *A Framework for Dynamic Subversion*, Master's Thesis. Naval Postgraduate School. Monterey, California. June 2003.

[SCS77] Schroeder, M.D., Clark, D.D., and Saltzer, J.H., "The Multics Kernel Design Project." *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pp. 43-56*,* November 1977.

[SCH75] Schiller, W. L., "The Design and Specification of a Security Kernel for the PDP-11/45," Mitre Techreport, MTR-2934, The MITRE Corporation, Bedford, MA 01730, March 1975.

[SCH83] Schell, R.R., "A Security Kernel for a Multiprocessor Microcomputer," *Computer*, Vol. 16, No. 7, pp. 47-53, July 1983.

[SCH85] Schell, R.R., Tao, T.F., and Heckman, M., "Designing the GEMSOS Security Kernel for Security and Performance," *Proceedings of the 8th National Computer Security Conference*, September 30 –October 3, 1985, Gaithersburg, MD.

[SCH03] Schell, R.R., Private correspondence, May 2003.

[SHO88] Shockley, W. R., "Implementing the Clark Wilson integrity policy using current technology," *Proceedings of the National Computer Security Conference*, pp. 29-36, 1988.

[SHIR81] Shirley, L.J. and Schell, R.R., "Mechanism Sufficiency Validation by Assignment," *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 26-32, April 1981.

[SIL83] Silverman, J.M, "Reflections on the Verification of the Security of an Operating System Kernel," *Communications of the ACM*, pp. 143-154, 1983.

[SMIDGE] Smidgeon Soft. *PE Browse Interactive*. http://www.smidgeonsoft.com, April 29, 2003.

[SOL00] Solomon, D.A., and Russinovich, M.E., *Inside Windows 2000,* Third Edition, Microsoft Press, Redmond, WA, 2000.

[SS72] Schroeder M.D., and Saltzer, J.H., "A Hardware Architecture for Implementing Protection Rings," *Communications of the ACM*, Vol. 15, No. 3, pp.157-170, March 1972.

[TEXT] Helios Software Solutions. "TextPad, the Text Editor for Windows," 2003, http://www.textpad.com, June 10, 2003.

[THI02] Thibodeau, P. and Verton, D., "Feds raid Mass. software firm suspected of ties to Al-Qaeda," *Computerworld*, December 6, 2002, http://computerworld.com/industrytopics/energy/story/0,10801,76487,00.html, March 3, 2003

[THO84] Thompson, K., "Reflections on Trusting Trust," *Communications of the ACM*, Vol. 27, No. 8, pp. 761-763, August 1984.

[THO89] Thompson, K., "On Trusting Trust," *Unix Review*, Vol. 7, No. 11, pp. 70-74, November 1989.

[THU01] Thurrot, P., "Tales of the Bizarre: Al Qaeda Allegedly Hacked Microsoft,"*Wininfo,* December 18, 2001. http://www.wininformant.com/Articles/Index.cfm?ArticleID=23535, March 3, 2003.

[VER01] VeriSign, Inc., "VeriSign Security Alert Fraud Detected in Authenticode Signing Certificates," March 22, 2001, www.verisign.com/developer/notice/authenticode/, March 3, 2003.

[VER03] Verton, D., "Terrorist probe hobbles Ptech," *Computerworld*, January 17, 2003, http://www.computerworld.com/securitytopics/security/story/0,10801,77680,00.html, March 3, 2003.

[WEB1] Weber, C., "Using IPSec in Windows 2000 and XP, Part One," *Security Focus,* December 5, 2001, http://www.securityfocus.com/infocus/1519, April 4, 2003.

[WEB2] Weber, C., "Using IPSec in Windows 2000 and XP, Part Two," *SecurityFocus,* December 20, 2001. http://www.securityfocus.com/infocus/1526, April 4, 2003.

[WEB3] Weber, C., "Using IPSec in Windows 2000 and XP, Part Three," *Security Focus,* January 2, 2002. http://www.securityfocus.com/infocus/1528, April 4, 2003.

[W2KRK] Microsoft Corporation, "Windows 2000 Resource Kit," 2003, http://www.microsoft.com/windows2000/techinfo/reskit/default.asp, April 28, 2003.

[XPE1] Microsoft Corporation, "Windows XP Embedded Home Page," 2003 http://www.microsoft.com/windows/Embedded/xp/default.asp, April 28, 2003.

[XPE2] Microsoft Corporation, "Windows XP Embedded Product Documentation," 2003, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/xpehelp/html/startpage.asp, April 28, 2003.

[XPE3] Microsoft Corporation, "Microsoft Windows Embedded Studio Development Tools for Windows XP Embedded," October 2002 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnxpesp1/html/xpeoverview.asp,May 5, 2003.

[XTS99] Final Evaluation Report Wang XTS-300. April 27, 1999. www.radium.ncsc.mil/tpep/library/fers/CSC-EPL-92-003-C.ps, June 9, 2003.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Dr. Ernest McDuffie
   National Science Foundation
   Arlington, VA

4. David Ladd
   Microsoft Corporation
   Redmond, WA

5. Andy Allred
   Microsoft Corporation
   Redmond, WA

6. Andy Newall
   Microsoft Corporation
   Redmond, WA

7. Jeana Jorgensen
   Microsoft Corporation
   Redmond, WA

8. Steve Lipner
   Microsoft Corporation
   Redmond, WA

9. Marcus Peinado
   Microsoft Corporation
   Redmond, WA

10. Marshall Potter
    Federal Aviation Administration
    Washington, DC

11. Ernest Lucier
    Federal Aviation Administration
    Washington, DC

12. Cynthia Irvine
Naval Postgraduate School
Monterey, CA

13. Roger Schell
Aesec Corporation
Pacific Grove, CA

14. Jessica Murray
Civilian, Naval Postgraduate School
Monterey, CA