INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

The Generic Policy Engine

by

Eugen Mate Bačić

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
March 6, 1998

© copyright
1998, Eugen Mate Bačić

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-57754-6

Canada

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis

The Generic Policy Engine

submitted by Eugen Mate Bačić, B.C.S
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University
May 5, 1998

Abstract

A generic system for the design of security policies is presented. The software system, called *Idyllic*, can be viewed as a *Generic Policy Engine* (GPE) capable of creating security policies identical to those found in many commercial, secure products but in a portable, more readable, succinct style, with comparable or greater trust, and verifiably correct. The policies are implemented using a verifiable, Scheme-like language, which generates security policies ranging from the classical, hierarchical models to the modern, commercial models. Idyllic provides system designers with well-known security entry points, a generic definition of “object”, and a means to manipulate these objects in terms of a security policy. The centralized nature of Idyllic allows for experimentation with different security policies rapidly and economically. Its verifiable nature ensures that Idyllic is capable of handling a majority of the security tasks of an operating system, network, or application and of being evaluatable to the highest security criteria possible.

Acknowledgments

I dedicate this thesis to my wife, children, and parents without whose continuous support, encouragement, and faith none of this would have been possible. Their enterprising spirit and constant attempts to keep me grounded and headed in the right direction does not go unappreciated. I only hope that I can return the support, kindness, and love in the future.

I wish to thank my supervisor, John Neilson, who persevered with me through three arduous years wherein I worked towards completing my thesis. His understanding during this period is much appreciated and his quiet belief and constant encouragement helped me achieve something that, at times, felt never ending.

I am also grateful to Dave Thomas of OTI who pointed out all the pitfalls along my chosen route, but over which I would stumble none-the-less. His gentle prodding over the past three years has not gone unnoticed and had I acted rather than listened to his advice, my journey would have been much easier.

Special thanks to the computer security community that has listened patiently to my treatises on the GPE notion and who have remained faithful.

Table of Contents

CHAPTER 1 INTRODUCTION	1
1.1 A NEW APPROACH	3
1.2 GOALS.....	5
1.3 STRUCTURE OF THESIS	6
CHAPTER 2 BACKGROUND.....	8
2.1 GENERAL CONCEPTS	9
2.1.1 <i>System and Physical Security</i>	9
2.1.2 <i>The System Boundary and Security Perimeter</i>	10
2.1.3 <i>Users and Trust</i>	11
2.1.4 <i>Trusted Systems</i>	12
2.1.5 <i>Subjects, Objects, & Processes</i>	13
2.1.6 <i>Security Policy</i>	14
2.2 DESIGN ELEMENTS	16
2.2.1 <i>System Structures</i>	16
2.2.2 <i>Identification and Authentication</i>	17
2.2.3 <i>Audit</i>	17
2.2.4 <i>Object Reuse</i>	17
2.2.5 <i>Trusted Path</i>	18
2.2.6 <i>The Trusted Computing Base</i>	18
2.2.7 <i>The Reference Monitor</i>	19
2.3 EVALUATING A SYSTEM'S TRUSTWORTHINESS.....	21
2.3.1 <i>Evaluation Criteria</i>	21
2.3.2 <i>Assurance</i>	22
2.3.3 <i>Functionality Requirements</i>	23
2.4 SECURITY POLICIES	24
2.4.1 <i>Transitions</i>	25
2.4.2 <i>Information Flow</i>	26
2.4.3 <i>The Entity Space</i>	26
2.4.4 <i>Entity Reclamation and Reuse</i>	27
2.4.5 <i>Access Controls</i>	27

2.5	SECURITY POLICY MODELS	29
2.5.1	<i>Bell-LaPadula Security Policy Model</i>	29
2.5.2	<i>The Biba Security Policy Model</i>	32
2.5.3	<i>Clark-Wilson Security Policy Model</i>	34
2.5.4	<i>Implementation of Security Models</i>	36
2.6	COMPARISON WITH OTHER WORK	36
CHAPTER 3 THE GENERIC POLICY ENGINE		41
3.1	THE ENTITY	44
3.2	IDYLLIC	48
3.2.1	<i>Why Scheme</i>	49
3.2.2	<i>Booleans</i>	52
3.2.3	<i>Equivalence Predicate</i>	53
3.2.4	<i>Pairs and Lists</i>	54
3.2.5	<i>Numbers</i>	55
3.2.6	<i>Control Structures</i>	57
3.3	DESIGN	58
3.3.1	<i>The Interface</i>	59
3.3.2	<i>Identification and Authentication</i>	61
3.3.3	<i>Audit</i>	62
3.3.4	<i>Access Control</i>	62
3.3.5	<i>Object Reuse</i>	63
3.3.6	<i>Trusted Path</i>	64
3.4	IMPLEMENTATION	64
3.4.1	<i>The Active Data Structure</i>	64
3.4.2	<i>Entity Based</i>	71
3.4.3	<i>The GPE Entity Definition</i>	72
3.4.4	<i>Trusted Computing Base</i>	75
3.4.5	<i>Reference Monitor</i>	76
3.5	THE ENTITY APPLICATION PROGRAMMING INTERFACES	78
3.5.1	<i>Instantiating a Copy of the GPE</i>	78
3.5.2	<i>The Unique Identifier</i>	79
3.5.3	<i>The Entity API</i>	79
3.5.4	<i>Identification and Authentication API</i>	80
3.5.5	<i>Audit API</i>	81
3.5.6	<i>Privileges API</i>	84
3.5.7	<i>Access Controls API</i>	85

3.5.8	<i>Security Policy API</i>	86
3.5.9	<i>Security Policy Management</i>	89
3.5.10	<i>Invocation Guideline</i>	90
3.5.11	<i>Housekeeping Functions</i>	91
3.5.12	<i>Object Reuse</i>	91
3.5.13	<i>Trusted Path</i>	92
3.6	TARGET PLATFORMS	92
3.7	TESTING	93
CHAPTER 4 DEMONSTRATION & EVALUATION		95
4.1	SYNTACTIC SUGAR.....	95
4.2	THE BELL-LAPADULA SECURITY POLICY MODEL.....	96
4.2.1	<i>Graphic Representation of Bell-LaPadula</i>	98
4.2.2	<i>Entity Creation and Destruction</i>	102
4.2.3	<i>The Least Privilege Principle</i>	103
4.2.4	<i>Mandatory Access Controls</i>	104
4.2.5	<i>Discretionary Access Controls</i>	107
4.2.6	<i>Object Reuse</i>	109
4.2.7	<i>Identification and Authentication</i>	110
4.2.8	<i>Trusted Path</i>	111
4.2.9	<i>Audit Mechanisms</i>	111
4.2.10	<i>The Security Policy</i>	112
4.2.11	<i>Using the Bell-LaPadula Security Policy</i>	114
4.3	MESSAGE TRUSTED GUARD	115
4.3.1	<i>Entity Creation and Destruction</i>	116
4.3.2	<i>Transmission Rights</i>	116
4.3.3	<i>Object Reuse</i>	117
4.3.4	<i>Identification and Authentication</i>	118
4.3.5	<i>Trusted Path</i>	118
4.3.6	<i>Audit Mechanisms</i>	118
4.3.7	<i>The Security Policy</i>	118
4.3.8	<i>Using the Message Trusted Guard</i>	119
4.4	FLEXIBILITY ASSESSMENT.....	120
4.5	PERFORMANCE CONSIDERATIONS	121

CHAPTER 5 CONCLUDING REMARKS	123
5.1 CONTRIBUTIONS	124
5.2 FUTURE WORK	124
5.3 SUMMARY	127
BIBLIOGRAPHY	128
APPENDIX A SECURITY FUNCTIONALITY	136
A.1 SECURITY CRITERIA	136
A.2 FUNCTIONALITY SERVICES	137
APPENDIX B ACRONYMS & GLOSSARY	140
APPENDIX C GPE: THE CODE	148
C.1 THE ENTITY DEFINITION	148
C.2 CONFIGURING, STARTING, & STOPPING THE GPE	157
C.3 ENTITY CREATION & CONTROL	158
C.4 IDENTIFICATION & AUTHENTICATION API	160
C.5 AUDIT API	161
C.6 PRIVILEGES API	164
C.7 ACCESS CONTROLS API	165
APPENDIX D EXAMPLES: THE CODE	167
D.1 BELL-LAPADULA SECURITY POLICY MODEL	167
D.2 MESSAGE TRUSTED GUARD	169

Table of Figures

Figure 1. Stylized Diagram of the Generic Policy Engine	3
Figure 2. System Boundary and Security Perimeter	10
Figure 3. Reference Monitor Model	20
Figure 4. Key Requirements of Trustworthiness (Assurance).....	22
Figure 5. Transitions of an Entity from Passive to Active or Process.....	25
Figure 6. Information Flow between Active and Passive Entities via a Process.....	26
Figure 7. Example Access Controls and Privileges in a Security Policy	27
Figure 8. Simple Access Control (or Protection) Matrix	28
Figure 9. Redefinition of Privileges as Access Controls	29
Figure 10. Bell-LaPadula Security Lattice	30
Figure 11. Simple Security Property & *-Property	32
Figure 12. Certification (Ci) and Enforcement (Ei) Rules	36
Figure 13. Generic Policy Engine Overview	42
Figure 14 . Security Attributes of an Entity.....	45
Figure 15. Classical Subject/Object Interaction as per TCSEC	59
Figure 16. Generic Policy Engine Overview	60
Figure 17. Pseudo-code Data Structure for GPE Entity	65
Figure 18. Idyllic Coded Data Structure for Entity	67
Figure 19. The Default Security Policy for the GPE	68
Figure 20. Data Structure for Entity with Expanded Association Lists	70
Figure 21. General Structure of an Entity.....	73
Figure 22. Generic Definition of an Entity.....	74
Figure 23. Instantiating a copy of the GPE.....	79
Figure 24. The Entity API	79
Figure 25. Identification and Authentication API	80
Figure 26. Defined Audit levels In the GPE.....	82
Figure 27. Audit Log API (Global)	83

Figure 28. Audit Log API (Entity Specific)	84
Figure 29. Privileges API	85
Figure 30. Access Controls API	86
Figure 31. GPE Security Policy API	87
Figure 32. GPE API Commands	87
Figure 33. Enumerated GPE Security Policy Function Calls	88
Figure 34. GPE Security Policy Management Private API	89
Figure 35. GPE Field List	90
Figure 36. Bell-LaPadula Security Properties	96
Figure 37. Bell-LaPadula Axioms	97
Figure 38. Bell-LaPadula Hierarchy Model Graphically Illustrated	99
Figure 39. Hasse Diagram of Two Objects	99
Figure 40. Hasse Diagram of Four Objects	100
Figure 41. Hasse Diagram of Eight Entities	101
Figure 42. Dual Requirement for Object Reuse at the System and GPE Level.	110
Figure 43. Bell-LaPadula Security Policy Model in the GPE	112
Figure 44. Updating the Access Controls via the GPE	114
Figure 45. Examining Access Controls via the GPE	115
Figure 46. Access Control Mediation by the GPE	115
Figure 47. Message Trusted Guard Security Policy Model in the GPE	119
Figure 48. Biba Security Policy Model in the GPE	121

Chapter 1

Introduction

Security policy refers to the rules governing the flow of information between two entities via predefined means and channels. A *policy engine* is a program that defines in logic those governing rules for a particular security policy, typically located within a trusted operating system. This thesis presents a new method for the generation of policy engines using a high-level, verifiable language and generic security functionality.

Conceptually, security policies are accomplished in two stages, *analysis* followed by *synthesis*. In the first stage, a natural language description of the security policy model is created so that it can be verified as reflecting the policy for the product being developed. The second stage describes this security policy model as a contrivance of logic within the actual product. Thus security policies rely on the correct modeling of the policies of the target environment and the ability of the designers and implementers to translate these models into functioning security policies within the product.

Historically the development of security policies has tended towards meeting the perceived needs of the largest market and embedding a static security policy within the product, typically an operating system. The first such system was Multics [44] which implemented the Bell-LaPadula Security Policy Model [15] which closely reflected the requirements of the military establishment [54]: the implementers. As time progressed and security became a serious concern to commercial interests, the security policies found within existing operating systems did not adequately reflect their needs. Further aggravation was wrought by the fact that the embedded security policies could not be

modified. In fact, the system was meant to handle information security in but one manner and if the methodology did not match that of the purchaser, the client was forced to either adjust their policies to more closely match that of the system or operate with less security than would be desired. To further aggravate the issue, even when two systems implement identical security policy models, say Bell-LaPadula, the implementations of these models within the products security policies don't match and communication between two machines often leads to security breaches.

In order to rectify this situation, the solution lies in approaching the situation in a manner similar to the compiler world. Security Policies should be guided by formal definitions using a verifiable language consisting of formal descriptions of the syntax and semantics. Besides eliminating the requirement to custom develop security policies for every product under development, formal descriptions of security policies by means of a *language* allows us to consider the development of a semantics-directed security policy generator, one which simplifies formal proofs of the validity of the security policies and ensures valid generation of the security policies for inclusion within a product.

Figure 1 illustrates such a system that produces security policies from formal definitions of the security policy model using a high order, lambda calculus based language. This provides several advantages. First, it creates a uniform, universal language for developing and proving the correctness of security policies and their models. Instead of the current assortment of policy verification mechanisms developers would have one language, one semantics, and one syntax to implement their security policies. Second, the proofs for the security policies could be automatically generated from the formal semantics of the base lambda calculus. Third, by providing a uniform language environment, connectivity between security policies can be improved since they would be based on higher level semantics than is currently available.

Figure 1. Stylized Diagram of the Generic Policy Engine

Several methods exist for formally describing the semantics of programming languages, and hence of security policies. Denotational Semantics [45, 52, 51] has gained particularly wide acceptance among semanticists and language designers. This thesis bases the verifiability of the language defined on the provability of systems developed using Denotational Semantics, such as VLISP [29].

1.1 A New Approach

Security policies can be developed more efficiently, in a greater variety, and with comparable or greater trust than with traditional implementations. Security policies

developed with the Generic Policy Engine (GPE) can be customized to reflect end-user security requirements as opposed to single-purpose security policies in common usage today. In order to provide for a *generic* policy engine a general, information neutral architecture must be used, one that is applicable across all operating systems or applications (henceforth collectively called “products”). Information must be encapsulated by the security measures required. The granularity of the information encapsulated defines the granularity of the security provided. The information thus encapsulated is the entity against which the GPE operates. By removing the restriction of specific products or data storage technologies, the GPE provides a trusted, extensible mechanism by which to implement security not only on a local system but also across heterogeneous networks, distributed and otherwise. Providing custom security policies that provide equivalent trust to those currently in use and with similar efficiency would be a major advancement in the information security field. To attain this level of flexibility with the *Generic Policy Engine* one must have at one’s disposal a high-level, verifiable, semantic driven language. All security relevant architecture must be written using this language which provides both the formalisms required for high security systems as well as the implementation specifics necessary to integrate it into a viable product. This thesis discusses such a language, *Idyllic* and its use as the base language of the *Generic Policy Engine*.

Idyllic is a Scheme-based [43] dialect. The *Generic Policy Engine* is the language plus the associated security extensions and functionality coupled with the ability to generate executable security policies. Thus, *Idyllic* provides a mechanism to easily define and implement verifiable security policies. By expanding upon the denotational semantics [28, 49] work on LISP-like languages performed by MITRE’s on VLISP [29], MIT on Scheme [19, 23, 24, 43, 48], Peter Lee [38, 40], and Apple on Dylan [46] and by means of well understood rule rewriting techniques, a verifiable system can be developed.

By utilizing Scheme-48 [35] as the base system, the prior work of Guttman on VLISP [29] can be extended to provide a denotational semantics for the Generic Policy Engine. In turn, a formal verification could be performed following the work of Guttman et al [29] or by utilizing some of the automated provers by Boyer [18]. This thesis utilizes a subset of Scheme 48 to model of the security engine in order to allow someone at a future date to leverage Scheme 48's denotational semantics and existing formal verification work.

By adopting an entity-based approach to the name space being secured we can further leverage the advantages of encapsulation and, to some extent, inheritance to create a succinct language definition for the creation of security policies. The new approach is to utilize a language renown for its small footprint, efficiency, and simplicity and use it to implement security policies. Embedding the GPE as a secure extension with a system would provide a trusted, extensible mechanism by which to implement security not only locally but across networks, both distributed or otherwise, within operating systems or applications, such as Web servers. The Generic Policy Engine would provide system and application developers means by which to provide equivalent trust to systems currently in use with smaller footprints and with at least equivalent verifiability, speed, and efficiency.

1.2 Goals

The goals of this thesis are to show that security policies can be developed more efficiently, in a greater variety, and with comparable or greater trust than traditional implementations. Security policies developed with the Generic Policy Engine can be customized to reflect end-user security requirements as opposed to single-purpose security policies in common usage today utilizing an information neutral architecture applicable across operating systems and applications. As proof of concept, a working Bell-LaPadula security policy model has been implemented as has a Message Trusted Guard.

Research on generic or multiple security policies and formal frameworks for such systems is examined as is the applicability of compiler technologies and their formalisms to the Generic Policy Engine in order to create a robust, verifiable meta-language. Although the Generic Policy Engine is designed to meet a high level of trust in security terms, and the security features provided by the Generic Policy Engine are outlined, a formal evaluation of the Generic Policy Engine is beyond the scope of this thesis.

1.3 Structure of Thesis

In Chapter 2 background information on information security fundamentals is provided, as is an overview of functional languages.

Chapter 3 provides a detailed description of Idyllic, the Generic Policy Engine, design and implementation issues, semantics, and an informal description of its execution. Proof of concept testing is also provided as is a comparison of common security functionality found within various commercial and publicly available security products along side similar ones developed using the Generic Policy Engine.

Chapter 4 examines a detailed case study of using the Generic Policy Engine to create security policies by implementing portions of the Bell-LaPadula Security Policy Model and a Message Trusted Guard – which is a portion of many commercial firewalls. The chapter ends with an examination of the flexibility of the implementation and performance considerations.

Finally, Chapter 5 summarizes the contributions of this research and compares it to related work. It discusses some of the limitations of the current methodology for creating Generic Policy Engines and propose several directions for future research.

Definitions of the terms and acronyms in common usage within the computer security field and this thesis are provided in Appendix B.

Chapter 2

Background

Prior to the mid-1980s, secure information systems were designed almost exclusively for military purposes. As computer systems have become increasingly used for storage and transportation of information across local and wide area networks, access to and protection of data has become a much wider concern. This concern has spilled over into the design of new information systems that are designed with various levels of security. Special purpose software is also being developed and deployed to guard against improper use of computer systems. Where only a decade ago only one security policy existed, today several policies and mechanisms have been proposed to defend operating systems and computer networks against malicious attack.

Standardizing security policies has been attempted in Canada, the United States, and Europe via various evaluation criteria, to varying degrees of success. The most recognized, and first, standard for the design of secure systems is the Trusted Computer System Evaluation Criteria [54]. The TCSEC defines several criteria for security, divided into three functions: security policy, accountability, and assurance. The Canadian Trusted Computer Product Evaluation Criteria [22], on the other hand, divides security into functionality services and assurance; the former is further divided into four security criteria: confidentiality, integrity, availability, and accountability.

Regardless of security criteria chosen, each describes a self-evident truth: any system can be considered secure if it respects a properly implemented and well-defined security policy. This has been reinforced with the release of the Common Criteria, a combination

of the best features of the TCSEC, CTCPEC, and Information Technology Security Criteria [33]. In order to simplify development efforts, many software developers choose to use the most widely available security policy: the Bell-LaPadula (BLP) model [15].

In order to maintain a clear focus on the contribution the Generic Policy Engine provides to the security community, and for the sake of brevity, this thesis distills out of each criteria the basic premise that security can be split into three functions: security policy, functionality, and assurance. Although the TCSEC, CTCPEC, and ITSEC weight each differently, for our purposes we must look upon all equally to give fair play to the notion of generalizing the creation and use of security policies. But prior to getting into the nuances of criteria and what is expected of any system being evaluated let's briefly review the fundamentals of computer security as it stands today.

2.1 General Concepts

This section introduces general concepts and terms used throughout the computer security field and apply to all applications. The concepts defined here will be used to show how the Generic Policy Engine meets the fundamental requirements for a secure application and how it can then be utilized to enhance security within all kinds of applications.

2.1.1 System and Physical Security

Security is typically divided into system security, that contained internally within the computer, and physical security, that obtained by physical means external to the computer [7, 25]. Both are valid and necessary areas of study in computer security. However, this thesis presents a method of enhancing the capabilities of system security to allow for greater controls over the movement of information within a system.

2.1.2 The System Boundary and Security Perimeter

A *system* is a computing environment within which system programmers have some level of control. Everything within a system is protected by the system and everything outside is not. This delineation between protected and unprotected entities is commonly called the *system boundary* and this boundary is the first line of defence against external threats to the system. Coherent security cannot be defined without an understanding of the threats.

Figure 2. System Boundary and Security Perimeter

The system boundary is fixed and requires a well defined interface which specifies exactly how external entities, such as users, terminals, other computers, peripherals, etc., interact with the system (see Figure 2). The system boundary offers protection against known and reasonable threats. It is unreasonable to expect a system to defend against lax

external security precautions, such as allowing passersby access to sensitive consoles or posting login information where it can readily be seen by anyone. Information and physical accesses to vital system information must be protected since the system is incapable of enforcing *physical security* upon the users and has no way of knowing if a particular user is valid or not.

Once a valid user has entered the system boundary, there are generally two types of entity at his disposal: security relevant and non-security relevant. The security relevant entities implement access controls, audit, and various other processes which are considered trusted to manipulate sensitive system information. In fact, this trust extends from this point down to the underlying hardware. The non-security relevant components and the security relevant components are separated by what is termed the *security perimeter*. The security perimeter creates a demarcation within which lie the hardware and security relevant software and data. Beyond the demarcation lie user programs, user data, peripherals, networks, printers, and other items the system controls and protects.

The information within the security perimeter must be tightly controlled and well-defined. Any breach within the security perimeter may well result in a cascade of security breaches defeating the entire system's security architecture. To minimize the likelihood of failure and to create well defined entry points into the security perimeter and known methods of information retrieval, storage, and manipulation, a well defined interface must be created and utilized.

2.1.3 Users and Trust

Most computer systems have users, individuals whose information the system utilizes and protects. The system, however, is not omniscient. It cannot determine whether *UserX* is the usual *UserX* who logs into the system or a poseur attempting to breach the system. The system attempts to ensure that information is only disclosed and modified by those who have appropriate access. The goal of a system is to have reasonable assurances that

an individual is who they say they are and that this trust is directly translated into actions associated with that individual's information. Notification of disclosure or modification beyond that normally allowed should be brought to the attention of the user by the system.

In order for a system to be reasonably certain the user requesting various accesses to a set of information is a valid user certain mechanisms need to be put in place. Typically a unique identifier, such as the user's name or an account number, must exist. Each unique identifier can be known by everyone but shared by no one. Although in a perfect world the unique identifier would be sufficient, in our world additional information is required to ensure that access is restricted to valid utilization of the various identifiers. To make a more robust authentication mechanism the user of the identifier must enter a password. This password is something known only to the user and thus is used as a key for entry into the system. Thus authentication is the process of providing to the system your unique identifier with something only *you know* or *have*. Modern authentication utilizes such diverse mechanisms as passwords, pass phrases, biometrics, retinal scans, smart cards, and personalized access cards. The first two are mechanisms utilize something the user knows, the next two something he is, and the last two something he owns. All are valid, but each offers differing levels of authentication strength.

2.1.4 Trusted Systems

Users must be trusted to protect data but the same does not hold true for computer programs that the users run. We thus face the dichotomy of trusting the users but not trusting their tools, namely the various computer programs which reside within the system they utilize. This dichotomy is resolved by understanding the basic edict of any secure system: *By default, access to information shall be denied unless explicitly granted*. Therefore, for us to allow a user to properly access information we must categorize the programs they utilize into one of three categories:

- Trusted:* Software that is fundamental to the correct operation of the system. This software is typically responsible for enforcing security and must operate flawlessly.
- Benign:* Software that is not responsible for any aspect of the system's security but is of known origin, perhaps supplied with the system being utilized. Being of known origin implies that it must be trusted (by the user) to manipulate protected information within the confines of the system's security. Any flaws are unintentional due to programmer error rather than malicious act and should not impact security.
- Malicious:* Software of unknown origin which, from a security perspective, must be treated as malicious and likely to attempt to breach the system's security.

In many secure systems the operating system is trusted while user programs and applications are not. This provides a built-in barrier strictly defining which elements of the system are trusted, keeping this list fixed for the duration of the system's life. In general "trusted software" has been written by individuals whom we can "trust", trust in the sense of creating software which will perform their designed duties and nothing more. Trusted software typically resides within the security perimeter, though "trusted applications", such as trusted databases, can reside outside the operating system and work, in effect, as microcosmic secure systems atop the secure system. Many modern secure systems do not allow linking into a security applications programming interface (API) to utilize the underlying features to guarantee the security of the system.

2.1.5 Subjects, Objects, & Processes

The functioning of a system can be viewed as successive operations on entities. Entities are all those things controlled and managed by the system. They can be files, peripherals, devices, physical memory, virtual memory, network packets, users, or programs –

anything that can hold data. Entities have historically been defined as being in one of two states: active or passive [15]. Active entities can access or manipulate passive entities and are called *subjects*. Passive entities, those being acted upon, are known as *objects*. Within most systems a subject is a process, job, or task operating on behalf of (i.e., a surrogate for) a user. Within this nomenclature, peripherals and daemons are treated as either subjects or objects depending on the observer's point of view [15, 25]. In order to alleviate confusion, the term entity (at times passive entity) shall be used to denote what is commonly referred to in computer security as an object. Subjects will be referred to as active entities. Processes will refer to those actions an active entity attempts against a passive entity.

In order to properly identify entities within the system all must maintain unique identifiers to maintain security.

2.1.6 Security Policy

The security policy defines the functioning of the security within the system. It must be defined in terms of *active entities*, *processes*, and *passive entities*. These entities must reflect the realities of the computer system and define all those pieces of the system that will be protected. The declaration of a precise set of rules governing authorization to the system and of access to entities is known as the security policy. Everything is fundamentally based on the utilization of unique identifiers for all entities. Historically many security policies were ad hoc collections of rules, many times conflicting, spread throughout the system code which were collected within the system over years of use. This lack of a clear security policy results in many of the system flaws and breaches seen today.

It is the security policy which determines the methods and mechanics of the access controls placed on the information, be they controls against disclosure, to ensure proper

handling, manipulation, and modification, or otherwise. Regardless, access control consists of three tasks:

- *Authorization*: determining which users are entitled to have access to which objects and by which processes the access is to take place.
- *Access Controls*: the rules that govern what level of access is to be granted to the user and the invoked processes.
- *Enforcement*: the act of ensuring the above tasks are *always* performed.

Access controls come in many varieties, both to control disclosure of information and manage manipulation. Some of the forms more readily found in the literature include:

- *Classic (Access) controls* where each entity explicitly lists which active entities and processes may access the data.
- *Role based* controls whereby active entities are grouped by roles and access to passive entities and processes is restricted based on these roles;
- *Capability based* controls which utilize tokens defining specific capabilities an active entity allowed to invoke. Each passive entity is tagged with various capabilities, all of which the active entity and processes must have prior to access being granted.

There is also the requirement to differentiate between access controls, which are defined and updated within the various entities, and the exercising of the controls, which occurs when a user attempts to access an entity. In other words, the former is the passive definition of rules governing any future accesses while the latter is the active attempt of a user to access information by providing sufficient evidence that it has the appropriate rights.

2.2 Design Elements

We will now examine the basic design elements for a trusted system including the methods employed to ensure the trusted system, once developed, can truly be trusted to perform its designed duties.

2.2.1 System Structures

Computer security views computer systems as one of two types: single machine, non-networked architectures and multiple machine, heterogenously networked distributed architectures. The single machine with users logging into it is the simplest to secure. There are no physically separate components operating far from the physical location of the system in question. The cooperation among processes is restricted to the current host greatly simplifying security since the issues of transport across networks need not be considered and the sharing of information is well contained. However, today the single, non-networked computer is disappearing quickly.

With the advance of the networked computer, with the requirement to share information and processing power across physically separate locales, and the worry about information sensitivity, archaic “grown” security can no longer suffice. As two or more disparate computer systems are linked, more often than not their security policies clash and security actually diminishes. With the policies having evolved over time, the ability to rectify the problems are exacerbated. To deal with the utilization of legacy architectures computer security experts have resorted to using encryption to at least guarantee the information gets safely to its destination. Encryption, however, is a coarse mechanism to use to protect information from disclosure. With no standardized method of creating security policies, encryption is becoming the de facto method of protecting information network wide, even though its utilization surrenders the fine grained control over individual information pieces that was available prior to the networking of the computers.

2.2.2 Identification and Authentication

Mechanisms must be put in place which provide for unique identification of every individual utilizing a system with the proviso that the system require each individual to authenticate themselves. The most common mechanism currently utilized, but by no means the only viable mechanism, is the standard login-password pair.

2.2.3 Audit

No matter the effort expended, it is never possible to declare a system 100% error free nor 100% secure. The small amounts of doubt that remain after a system is complete must be removed in order to ensure that unforeseen attacks will be recorded enabling system personnel to correct the breach and, perhaps, apprehend the culprit. The method employed in trusted systems is called audit and it is the systematic recording of a given set of events into a protected data store by the system. For example, under UNIX it might be reasonable to expect the auditable events to include file access attempts, system clock resets, file reclassification, file execution, process instantiation and exit, program exit status, process forks, file open, file close, file location movement, access to specific peripherals, specific file access attempts (i.e., su), etc. Each time an auditable event is triggered, the system writes relevant information, such as the individual performing the action or request, the date and time, the status of the request and other pertinent information to a log file, typically stored and protected separate from the system's regular files. This log file must be regularly examined to determine the state of the system and ensure that no breaches have occurred.

2.2.4 Object Reuse

A system's resources are finite, and as such must be reused in order for the system to be at all usable. In a secure system the reuse issue means ensuring that information to which a given individual is not privy should not be provided accidentally or otherwise. With the dynamic makeup of systems, it become imperative that prior to reallocating reclaimed

storage space the space must be sanitized of all information it currently contains. One acceptable mechanism is the scrubbing of the memory by overwriting it with a pseudo-random binary sequence.

2.2.5 Trusted Path

At times it is required that either the user of a system or the system itself impart information of such sensitivity that it must not be interrupted or ignored by the system nor the user. When this type of communication is required, either the user or the system can invoke a trusted path whereby a secured communications link is established while all other processes are suspended pending confirmation. Trusted paths are invoked during information downgrades and login; A commonly used trusted path is found in Microsoft's Windows NT login where a trusted path is invoked via the control-alt-delete sequence instantiating the login screen; this screen was designed to be non-bypassable and in reality passes NSA and CSE trusted path requirements.

2.2.6 The Trusted Computing Base

A Trusted Computing Base (TCB) is the set of software, firmware, and hardware elements of a product involved in enforcing a product's security policy. Typically the system designer designates the portion of the product which is the TCB. This must include any piece of software, hardware, or firmware if manipulated by something outside the TCB boundary would result in a violation of the product's security policy. Anything that can manipulate data structures or software within the TCB must be considered *part of* the TCB. These elements are considered part of the TCB and are termed to be within the TCB boundary. The boundary is the delineation that designates which software, firmware, or hardware elements must be run in a trusted¹ mode and those that can be run in an untrusted mode. The security policy operates upon those software, firmware, and

¹ Trusted mode is typically invoked via hardware protected states and is commonly referred to as the privileged state: untrusted mode is typically known as the user state.

hardware elements *within* the TCB boundary. The boundary should include anything that can manipulate or is manipulated by the TCB and requires protection from outside interference.

2.2.7 The Reference Monitor

In 1972 Anderson [8] introduced a concept that continues to form the foundation of all trusted information systems: *The Reference Monitor*. The Reference Monitor model defines three guiding principles which, when implemented, provide for an ideal design to control information flow.

The function of the Reference Monitor is to validate all references (e.g., references to programs, data, and peripherals) made by programs in execution against those authorized for the subject (e.g., the user). The Reference Monitor not only is responsible for assuring that the references to access shared resource objects are authorized but also to assure that the reference is the right kind (i.e., read, or read or write, etc.).

The Reference Monitor model typically requires a combination of hardware, software, and firmware: this combination is called the *Reference Validation Mechanism*. The three guiding principles defined by Anderson were:

- *The Reference Validation Mechanism must be tamperproof.*
- *The Reference Validation Mechanism must always be invoked.*
- *The Reference Validation Mechanism must be small enough to be subjected to analysis and tests to ensure that it is correct.*

Figure 3. Reference Monitor Model

The operational characteristics of the reference monitor can be succinctly illustrated in Figure 3 which shows a reference monitor utilizing a security policy to allow or deny access requests from users based on a predefined set of rules associated with the policy. Trusted systems are designed to always invoke the reference monitor and to ensure it is not bypassable thereby ensuring information flows as illustrated in Figure 3. It is within the reference monitor that all requests to access information are resolved by means of the security policy.

With the development of the US Department of Defense's *Trusted Computer System Evaluation Criteria* (the Orange Book) it was determined that the above principles were insufficient and the notion of a TCB, as described above, encompassing all security-relevant functionality was required. As the Orange Book was oriented towards operating systems of the 1970s and early 1980s an effort must be exerted to apply these working principles to modern micro-kernel, object-based operating systems. The numerous advantages of object-based systems allow for a return to Anderson's purist view, perhaps forgoing the requirement for a TCB altogether.

2.3 Evaluating a System's Trustworthiness

As with all system development, secure system development follows well established procedures with supporting documentation. Typical system development follows the following steps:

- Requirements
- Specifications
- Design
- Implementation

Although many systems are now developed using rapid application development methods, they still generate documents which fall closely into the steps outlined above. To evaluate a system's trustworthiness the additional requirement of verification of transformation between steps is added. This verification, which can range from an informal examination to mathematical precision, ensures that the intended system is built and that it remains consistent from step to step. Further, the security policy, which is typically defined during the requirements phase, must be properly implemented and defined at each step and it is the system architect's duty to ensure its proper implementation. Although no steps are added to a typical system development process, rigour is introduced to ensure the system's development is as correct as humanly possible.

2.3.1 Evaluation Criteria

The purpose of security criteria is to specify a standard for the evaluation of security in a product. This standard establishes a common denominator for security as measured by three groups of people. First, the system user is given a metric against which to judge the level of trust that can be placed into a computer system which is to store and protect sensitive information. Second, vendors are provided with a set of criteria that must be met for a product to be deemed trusted and thus satisfy customer requirements. Finally, the

criteria provide purchasers with guidelines that can be used to specify the security needs for a particular computer product.

2.3.2 Assurance

Assurance in the various criteria is classified on a scale ranging from minimal protection (i.e., effective failure) to verified protection. In order for a security policy product be applicable across the widest range of trust levels it must be designed to be flexible. The criteria explicitly define the requirements for each level, each building upon the assurance requirements of the previous. Full details for each particular criteria can be found in the respective documents. The following figure defines the key requirements that must be evident within a system to comply with the design for a verifiable trusted product (i.e., T-6 in the CTCPEC or A1 in the TCSEC):

Discretionary Security Protection	Active and passive entities are distinguished allowing users to protect private information from unauthorized users.
Controlled Access Protection	Discretionary controls are implemented with a fine granularity providing for individual user accountability for their actions through login procedures and auditing.
Tagged Security Protection	Mandatory controls (at the discretion of the system) over active entities, passive entities, and processes must be present.
Structured Protection	A trusted computing base must be defined. The nucleus must be designed in a structured and modular way. The least-privilege principle must be enforced. A trusted path must be established between a user logging in and the system.
Security Domains	The policy enforced must be isolated from the policy mechanisms in a security nucleus.
Verified Design	The security model must be formally proven. Formal methods including design specification and verification must be used to prove that the trusted computing base is correctly implemented.

Figure 4. Key Requirements of Trustworthiness (Assurance)

2.3.3 *Functionality Requirements*

The requirements found within the three international criteria² define five fundamental requirements, which are briefly introduced in this section. The thesis is organized based upon these requirements.

Requirement 1: Security Policy

The product must enforce a well-defined security policy. The security policy operates upon the entities within the product and defines all access as that of an active entity accessing a passive entity via a known process. The security policy defines how active entities can access passive entities in the system and consists of at least one of the two types of controls: mandatory and discretionary.

Requirement 2: Tagging

Each entity controlled by the security policy must be tagged with a label. This label indicates information relevant to the security policy in determining how information can be accessed. In the case of the Bell-LaPadula model, the labels would indicate the clearance of the active entity and the sensitivity label of the passive entity.

Requirement 3: Identification & Authentication

The active entities within the product must be uniquely identified and authenticated and assigned relevant information pertaining to the security policy. In the case of the Bell-LaPadula model this information would be in the form of a clearance indicating what sensitivity of information the active entity would be able to see. In the case of role based

² Throughout this thesis we shall focus on the three more proven criteria: Canada's CTCPEC, the U.S.'s TCSEC, and the European ITSEC. Although the Common Criteria has been available at time of writing, it has not yet been proven through extensive case studies and evaluations done at the national level. Further, as the Common Criteria attempts to wed the three international criteria into a comprehensive whole, it is not illogical to assume that the thesis is equally applicable to the Common Criteria.

security policies, this information would be which roles a particular active entity was assigned.

Requirement 4: Accountability

Security relevant events such as entity accesses, entity modification, process instantiation, and product access by a particular user (logins) must be recorded in an audit trail. The purpose is to detect and diagnose malicious or non-malicious actions that lead the system to a non-secure state.

Requirement 5: Operational Assurance

The security software must be designed and implemented in such a way that all the preceding requirements are enforced. A suite of built-in tools must be provided to test the security features of the system during normal operations. The security aspects of a product must be of sufficiently simple organization and complexity to be subjected to and pass a wide spectrum of analysis and tests.

2.4 Security Policies

For a product to be deemed “trusted” it must enforce an explicit security policy. The security policy is a set of rules that regulate the use of information, including its processing, storage, distribution, and presentation. The security policy is typically defined in terms matching those outlined within one of the accepted security criteria (U.S. TCSEC, Canadian CTCPEC, European ITSEC, or the International Common Criteria). Within each criteria explicit levels of trust are defined with specific requirements with respect to the security policy. For a product to be deemed secure and trusted it must be specified and meet the requirements defined within the targeted assurance level. The security policy continuously protects against tampering and/or unauthorized change. No computer product can be considered secure if the basic software, hardware, and firmware

mechanisms that enforce the security policy are themselves subject to unauthorized modification or subversion.

2.4.1 Transitions

For a security policy to operate efficiently it must be able to isolate entities within its control, to guarantee the mediation of access requests, and ensure that controlled and noncircumventable audit exists to track information flows. All security functionality falls within the bounds of one or more of isolation, mediation, or audit. Nomenclature varies between security criteria, but for our purposes we can term everything under the control of the Trusted Computing Base (TCB) to be an entity³. In most products, entities are in one of three distinct states: active, process, or passive (see Figure 5).

Figure 5. Transitions of an Entity from Passive to Active or Process

As an entity migrates from one state to another, a change of context occurs. The entities remain manipulatable, though restrictions are placed by the security policy as to which entities can be manipulated. Although an entity may appear to be in a process state in one domain, within another it may be viewed as a passive entity against which an operation

³ This is the agreed upon definition of both the Canadian Trusted Computer Product Evaluation Criteria and the Harmonized Common Criteria, although the two Criteria utilize the term "object".

can be performed. This allows one process entity to manipulate another process entity, or even another user entity, since they remain "passive" from the viewpoint of the manipulating entity.

2.4.2 Information Flow

In all computer systems information flows in one of two directions: from the active entity to the passive entity, or from the passive entity to the active entity. The paths taken can vary as can the input and output media. Information security is, therefore, defined as the proper flow of information between a passive entity and an active entity via a number of processes. The information is tightly coupled to the active entity so as to ensure that the active entity is authorized to access the information. The underlying principles that govern *information flow* are defined by the security policy.

2.4.3 The Entity Space

The entity space of a Trusted Computing Base, illustrated in Figure 6, contains all the entities under the control of the TCB. All entities transit between the three known states (Active, Process, and Passive), for example making a state transition from Passive to Active via an authenticated login. These state transitions allow entities to be associated with processes and as owners and/or manipulators of other entities.

Figure 6. Information Flow between Active and Passive Entities via a Process.

2.4.4 Entity Reclamation and Reuse

Entities represent actual resources being protected within an application or system. Whenever a user logs off the corresponding entity must revert to a passive entity. If, however, that user was transient (such as a batch job), it would have to be destroyed upon log off. The destruction and creation of entities is defined by the system designer and must not contravene any aspects of the security policy. All criteria require some form of entity reuse (i.e., garbage collection) ensuring that freed resources are not reallocated without first being sanitized. All creation and destruction must be performed by the TCB.

2.4.5 Access Controls

As indicated earlier, everything controlled by the Trusted Computing Base, and thus protected by the security policy, must be defined as an entity. The set of rules governing the interaction between entities is known as the security policy. Each entity is tagged with a unique identifier and with other information denoting its access controls and privileges. These tags, as illustrated in Figure 7, are used by the TCB for isolation, mediation, and audit and provide the flexibility of creating security policies that can be access control based, capability based, or any other secure methodology. No limit is placed on the number of tags that can be associated with an entity.

Entity	Unique ID	/home/eugen/thesis/gpe.doc
Access Controls	read	Eugen, Ariana
	write	Eugen, Goran
	execute	—
	copy	Eugen
	delete	Eugen

Figure 7. Example Access Controls and Privileges in a Security Policy

Access attempts are approved or rejected by the security policy via the reference monitor by examining the tags of the entities in question and determining the validity of the access request. Thus, if user *Ariana* attempted to write to the file “/home/ebacic/thesis/gpe.doc” she would be refused since her unique ID is not listed in the access control list for the given entity. Perhaps the following diagrams better illustrate the mechanism implied by access controls in a security policy.

		Entity 1	Entity 2	Entity 3
D o m a i n	User 1	read	read write	
	User 2		read execute	read write

Figure 8. Simple Access Control (or Protection) Matrix

Figure 8 illustrates a simple case where *User 1* has read privilege to *Entity 1*, and read and write privileges to *Entity 2*, and no privileges to *Entity 3*. *User 2*, on the other hand, has no privileges to *Entity 1*, but has read and execute to *Entity 2*, and read and write to *Entity 3*. Utilizing this table in a simple security policy in terms of accessing entities, we can easily determine what *User 1* and *User 2* have access to, and what form of access can be granted. These lattices or matrices are simple devices but can be used exceedingly efficiently in the implementation of security policies which can be translated into a lattice model, such as Bell-Lapadula.

By defining a few terms as privileges (e.g., read, write, and execute) we can create complex relationships in terms of *Subject* and *Object*, as defined by Bell and LaPadula [15]. For example, collapsing Figure 8 by column results in the more familiar lattice illustrated in Figure 9. Collapsing by column would have yielded a capabilities list [53].

Object 1	<i>Read</i> = (<i>User 1</i>) <i>Write</i> = () <i>Execute</i> = ()
Object 2	<i>Read</i> = (<i>User 1</i> , <i>User 2</i>) <i>Write</i> = (<i>User 1</i>) <i>Execute</i> = (<i>User 2</i>)
Object 3	<i>Read</i> = (<i>User 2</i>) <i>Write</i> = (<i>User 2</i>) <i>Execute</i> = ()

Figure 9. Redefinition of Privileges as Access Controls

Since we are creating a *Generic Policy Engine*, we must provide a generic mechanism of associating relevant tagging information with entities. In order to provide a uniform entity base upon which to operate each entity must be tagged in an identical manner within a fixed framework. This allows for a uniform implementation of the security policy, regardless of whether it is access control based or role based, privilege or capability based, disclosure or integrity based.

2.5 Security Policy Models

The formal definition of a governing rule set for the functioning of a trusted system is called a security policy model. These models can be formally provable via verification systems such as *Gypsy* [5], *Verdi/Eves* [21], and *Z* [17].

The following sections outline the most familiar security policy models currently understood, studied, and utilized in the security arena.

2.5.1 Bell-LaPadula Security Policy Model

Lattice models trace their ancestry back to the original work done by Bell and LaPadula [15] in 1975 for the Multics Project. Their seminal work outlined what has become

known as the Bell-LaPadula Model and defines mathematically the proper functioning of a hierarchical security system. The Bell-LaPadula Model was the first one to be provided with mathematical underpinnings as well as formalisms and a formal proof of correctness. It is most commonly used in situations where hierarchical information is processed, and thus is typically seen as a government or military security policy model.

		User Clearance		
		User 1	User 2	User 3
		(Top Secret)	(Unclassified)	(Secret)
Data Classification	Top Secret	read write	write	write
	Secret	read	write	read write
	Confidential	read	write	read
	Unclassified	read	read write	read

Figure 10. Bell-LaPadula Security Lattice

The lattice described by Bell and LaPadula, Figure 10, has user clearance along the top axis and data classification along the left axis. Reading down from User 2 to the intersection with Secret we notice that User 2 has write access to the Secret data classification. In other words, the user can write *up* to Secret. However, User 2 does not have read access. Conversely, User 3 can read and write to the Secret data classification.

Therefore, whenever the clearance of the user is equal to or greater than the data classification, the user has the right to read the information. If the clearance of the user is less than or equal to the data classification, the user can write the information.

Strictly interpreted, Bell-LaPadula’s lattice model allows users to write to classification levels equal or higher then their own and read to classification levels equal or lower.

The Bell-LaPadula Model is described in terms of a simple triple containing the active entity (commonly referred to as the *Subject*), the passive entity (commonly referred to as the *Object*), and the process. Two simple rules govern the Bell-LaPadula Model:

Simple Security Property: Also known as the no read up (NRU) rule, states that an active entity with security label L_s can only read information of a passive entity with security label L_o if and only if L_s dominates L_o .

*-Property: Also known as the no write down (NWD) rule, states that an active entity with security label L_s can only write information to a passive entity with security label L_o if and only if L_o dominates L_s .

More simply stated, the Simple Security Property says that a user at a given level, say *Secret*, cannot *read* information at a higher classification level, say *Top Secret*. This ensures that information does not flow downward. It does not restrict said user from reading below his classification level, say *Confidential*. The *-Property is the converse of the Simple Security Policy, and in many ways is in effect identical to it, but worded differently. Thus, the *-Property says that a user at the *Secret* level cannot write *Secret* information down to the *Unclassified* or *Confidential* levels. Figure 11 illustrates the two rules. These rules provide for reading and writing by an active entity to a given passive entity if and only if the active entity and passive entity are within identical domains, i.e., $L_s = L_o$.

Figure 11. Simple Security Property & *-Property

The Bell-LaPadula Model is defined as a *Confidentiality Model*.

2.5.2 The Biba Security Policy Model

In 1977 Biba [16] developed an integrity policy model that is the dual⁴ of the Bell-LaPadula model. In an attempt to show that an integrity-based model was feasible and provide a useful operating environment, Biba created his integrity model as the *inverse* of

⁴ In computer security the term "dual" is utilized to denote something that is effectively the opposite of something. In this case, Biba is the dual of Bell-LaPadula, or the opposite. The utilization of both the Biba and Bell-LaPadula security policies on a single machine would result in information being unable to move upwards or downwards!

Bell-LaPadula. Taking the dominance relationship found in the Bell-LaPadula model, as illustrated below, and inverting it resulted in what became the first Integrity Model.

Simple Security Property: An active entity with security label L_s can only write information of a passive entity with security label L_o if and only if L_s dominates L_o .

*-Property: An active entity with security label L_s can only read information to a passive entity with security label L_o if and only if L_o dominates L_s .

As can be seen, Biba's model replaces each occurrence of "read" in Bell-LaPadula's model with "write" and each instance of "write" with "read". This results in information having less integrity⁵ as information of a lower integrity is inserted. Bell-LaPadula's allows information that was less sensitive to be moved up to a higher sensitivity without problem, it was the downward motion that was restricted. Biba allowed the inverse, namely information with less integrity could not be moved up but information of a higher integrity could be brought downwards.

Similarly, where Bell-LaPadula allowed a higher sensitivity or classification label to read and incorporate information from a lower label, Biba was the exact opposite. This reversal meant that implementing both security policies would result in a system where all information stayed at a constant security level, unable to move up or down due to the conflicting security policies rules governing information flow. The Biba Model is defined as an *Integrity Model*.

⁵ The insertion of lower integrity data with higher integrity data, via the Biba model, results in information of a mixed integrity level which is somewhere between the lower and higher integrity levels.

2.5.3 Clark-Wilson Security Policy Model

Since Biba's model, a number of integrity models have surfaced. In 1987 Clark and Wilson presented their seminal work on commercial integrity and their model, since dubbed the Clark-Wilson Integrity Model [20]. Clark and Wilson's focus on *commercial* integrity was a major departure from most work being performed in the security arena at the time. Most security models were still focused on confidentiality and the few focused on integrity. Since Clark-Wilson work has accelerated not only on commercial integrity but on defence requirements, including efforts to link the Clark-Wilson model to the Bell-LaPadula model.

As efforts into integrity models accelerated, defining what integrity meant to the field of computer security became paramount. In 1989 during Invitational Workshop on Integrity Policy in Computer Information Systems [57] Wilson and Bacic formulated the following as a concise definition of what constitutes integrity within a computer system:

- A system maintains integrity when change occurs if and only if appropriate.
- One form of appropriate change is that which tracks real world expectations for completeness, accuracy, timeliness, consistency, interpretability, and/or relevance of the system and its data.
- A system remains consistent with an earlier state.

This definition was used to define the Integrity Criteria within the Canadian Criteria [22], for the research and papers on Process Execution Controls [9, 10, 56], and CSE position papers for subsequent WIPCIS conferences.

The most referenced of all the integrity models is the Clark-Wilson Model [20] which provided a definition of commercial integrity concerns to the computer security arena for the first time. Clark-Wilson use the notion of trusted processes managing the flow of information to and from various data stores. As information flows through the model it

must be approved for inclusion within a data store by a given or a set of given verification procedures. These verification procedures restrict all modifications to well-formed transactions on the data being performed by authorized users as defined by the principle of separation of duties. The well-formed transactions can further be restricted per user so as to ensure users can only manipulate the information in predefined ways.

Clark-Wilson's model does not have a formal proof associated with it nor is there an existing implementation of the full model as described in their paper. However, it has opened the door to numerous other papers on integrity, removing the requirement that security policies always address, at least to some extent, military concerns.

The model, in essence, is concerned with integrity of computer systems and consists of two types of rules: certification rule (Ci) and enforcement rules (Ei). Those rules, as defined by Clark and Wilson [20] are:

C1 All <i>Integrity Verification Procedures (IVPs)</i> must properly ensure that all <i>Constrained Data Items (CDIs)</i> are in a valid state at the time the <i>IVP</i> is run.	
C2 All <i>Transformation Procedures (TPs)</i> must be certified to be valid. That is, they must take a <i>CDI</i> to a valid final state, given that it is in a valid state to begin with. For each <i>TP</i>, and each set of <i>CDIs</i> that it may manipulate, the security officer must specify a "relation", which defines that execution. A relation is thus of the form: $[TP_i, (CDI_a, CDI_b, CDI_c, \dots)]$, where the list of <i>CDIs</i> defines a particular set of arguments for which the <i>TP</i> has been certified.	E1 The system must maintain the list of relations specified in rule <i>C2</i>, and must ensure that only manipulation of any <i>CDI</i> is by a <i>TP</i>, where the <i>TP</i> is operating on the <i>CDI</i> as specified in some of the integrity enforcement relations.
C3 The list of relations in <i>E2</i> must be certified to meet the segregation of duty requirement.	E2 The system must maintain a list of relations of the form: $[UserID, TP_i, (CDI_a, CDI_b, CDI_c, \dots)]$, which relates a user, a <i>TP</i>, and the data objects that <i>TP</i> may reference on behalf of that user. It must ensure that only executions described in one of the relations are performed.

C4 All <i>TPs</i> must be certified to write to an append-only <i>CDI</i> (the log) all information necessary to permit the nature of the operation to be reconstructed.	E3 The system must authenticate the identity of each user attempting to execute a <i>TP</i>.
C5 Any <i>TP</i> that takes an <i>Unconstrained Data Item (UDI)</i> as an input value must be certified to perform only valid transformations, or else no transformations, for any possible value of the <i>UDI</i>. The transformation should take the input form a <i>UDI</i> to a <i>CDI</i>, or the <i>UDI</i> is rejected. Typically, this is an edit program.	E4 Only the agent permitted to certify entities may change the list of such entities associated with a <i>CDI</i> and list of users associated with a <i>TP</i>. An agent that can certify an entity may not have any execute rights with respect to that entity.

Figure 12. Certification (*Ci*) and Enforcement (*Ei*) Rules

To date there are no commercial implementations of the Clark-Wilson model.

2.5.4 Implementation of Security Models

Numerous security models exist that apply to the security of information systems. Although it would be interesting to implement every known security policy utilizing the GPE, space, time, and brevity require that we focus on a few well known examples. For the purposes of this thesis only access control models will be implemented as they are the security policies most prevalent in the real world and with which most people are familiar. Two example security policies have been selected. The Bell-LaPadula Security Policy Model is an access control policies in common use world-wide. The Message Trusted Guard is security policy controlling electronic mail transmission commonly used in government deployed firewalls. Both are described in Chapter 4.

2.6 Comparison With Other Work

Much work has begun on the creation of metapolicies [30, 31] and TCB subsets [1, 2, 3, 36], Much of this work is similar in nature, namely the combination of existing security policies currently in the field into workable, functional, and understandable super policies. Current research on generalizing security policies is towards the combination of

disparate security policies with the assumption the interconnected whole will be comprehensible and more reliable due to the expended effort. The Generic Policy Engine approaches the issue of generalizing security policies from the opposite direction seeking to create a framework by which security policies can be created and that can interact due to the nature of the underlying infrastructure of the framework rather than on logic which intends to link existing policies together.

TCB Subsets

Abrahms et al [1, 2] most closely relates to the Generic Policy Engine's notion of well defined security policies which can be interconnected due to their framework. Abrahms examines the work by Shockley and Shell in 1987 [47] and their implementation of security policies against the TDI [55]. Abrahms et al introduce the notion of a hierarchy of TCB subsets, offering a more flexible framework based on a separation kernel. Each subset does a particular task and performs that task well. It relies on the framework to interconnect the various pieces much like nodes on a network are connected together and communicate.

In many ways, TCB Subsets look and behave similarly to the composition problem as examined by Lee et al [37] at the University of Toronto. However, TCB Subsets incorporate the metapolicy ideas of Hosmer and allow for each TCB Subset to be evaluated separately, with the metapolicy (and its accompanying rules) governing the interaction and underlying security offered by the combined system.

TCB subsets offer the possibility of "componentized" security software, each of which has a well defined interface definition (API) and each of which (typically) performs a single task. TCB subsets are typically presented by their authors as offering a solution to the problem of distributed operating systems and security. TCB Subsets further propose the creation of a set of access controls portable and consistent between all

implementations, namely the Access Control Decision Function, Access Control Enforcement Function, and the MetaPolicy Function. This stresses the requirement for conformity to simplify the later evaluation and verification, at the expense of flexibility of meeting business rule requirements. A major advantage, however, is that once the access controls have been evaluated once for any given system, they will hold true, due to the rigidity of implementation and definition, for all future implementations and evaluations.

Thus Abrahms' TCB Subsets provide for a mechanism to define a security topology. This topology would describe the interaction between various security components, restricting information flow in accordance with a given rule base, the Metapolicy Rules, and a set of access controls, the Access Control Rules. These rules would be user definable in terms of the underlying separation kernel, enforcement functions, decision functions and metapolicy function. Unfortunately, as indicated by Abrahms, they do not appear generic enough to implement more than specific of access control policies. In fact, Abrahms et al have formalized a definition model for access control policies; a criteria for their examination, evaluation, and description. To date, no implementation of the TCB Subset idea has been completed.

Generic Security APIs

The Generic Security Service Application Program Interface (GSSAPI) presented by Linn at the 1990 USENIX Security Workshop [42] presents an API which addresses specific concerns regarding the definition of security services. Linn's GSSAPI, currently an RFP out for Internet approval as a security API standard, assumes and addresses several basic goals, including:

- mechanism independence:
the GSSAPI is independent of the particular underlying mechanisms used to implement any given portion of the API;

- protocol environment independence:
no particular communications protocol is assumed; and
- suitability to a range of implementation placements:
no restrictions are placed upon the GSSAPI clients to reside within a given TCB perimeter.

The GSSAPI is network security specific and attempts to generalize the security requirements prevalent to networks available today. The GSSAPI provides peer entity authentication, per-message data origin authentication, and data integrity protection. A certain amount of data confidentiality is also provided; however, the GSSAPI is primarily a network message authenticator.

The Generic Policy Engine complements the GSSAPI by providing operating system level support for confidentiality, integrity, availability, and identification and authentication. The Generic Policy Engine can be extended to provide its services across a network, however nothing precludes the Generic Policy Engine from integrating the services of the GSSAPI. In fact, by integrating the GSSAPI's services within a fully networked Generic Policy Engine implementation, it is possible to create a highly trusted network operating system environment in which communications, or at the very least certificates⁶, are encrypted.

Metapolicies

In 1991 Hilary Hosmer presented the concept of metapolicies [30], or “policies about policies”; a compelling method of developing large, complex and interrelated trusted systems that can be used by all security conscience organizations. Hosmer’s metapolicies presented a framework for clarifying policies and for successfully coordinating security policies and subpolicies. By using the concept of metapolicies, Hosmer showed how to

⁶ Certificates are unique identifiers which are encoded in a unique fashion allowing the recipient to authenticate the validity of the message and identify the originator. Certificates are considered tamper resistant.

coordinate multiple security policies on the basis that a well-defined subpolicy structure existed.

As an example, Hosmer examined and described several metapolicies implicit within the Simple Security Policy of the Bell-LaPadula Security Policy Model, these included:

- Policy Description Metapolicy
- Policy Constraint Metapolicy
- Subpolicy Interaction Metapolicy
- Organization Control Metapolicy
- Automated Information System Metapolicy
- Multipolicy Coordination Metapolicy

The coordination of these policies, typically explicitly defined within hard coded security policies, were presented as coordinating components under the supervision of a metapolicy framework. The notion of cooperating policies which form a metapolicy is a form of decomposition of complex policies in order to more easily verify their correctness. Although still at the research stages, it would be constructive to see how the metapolicy philosophy works when the Generic Policy Engine is utilized to create each of the policies which compose the metapolicy being evaluated.

Chapter 3

The Generic Policy Engine

The Generic Policy Engine (GPE) allows programmers to create and generate a valid security policy from a high-level, verifiable semantic description of a security model. It must provide a one-for-one mapping between the entities to be protected in the calling application and the security information maintained by the GPE. In order to accomplish this, the GPE utilizes an entity-based approach defining an encapsulated entity-like entity with all the security relevant components within its boundary. Security relevant application programming interfaces (APIs) are provided so applications can manipulate security attributes and determine proper information flow, call the security policy, examine audit logs, and perform regular maintenance on the entity. The GPE's purpose is to provide sufficient security to an application so that the application requires only minor security specific instructions to meet even the highest security requirements of the various international evaluation criteria [22, 33, 54].

Figure 13 presents an overview of the Generic Policy Engine. The GPE is written in and controlled by the Idyllic, a symbolic, lambda calculus-based language. The language is fully extensible and comes with a highly capable library of functions. The GPE's primary data structure is called the entity containing all the security data pertaining to an entity under its control along with the application programming interface (API) to manipulate the entity. Customizable elements, which can differ from application to application, can be provided by application programmers and are referenced via the APIs. The two programmer configurable components are the I&A mechanism and the security policy.

The various components are discussed at lengths, in coming sections, as are the structure of the entity, its data elements, the facilities provided in the various APIs, the functionality provided by Idyllic, and the security functionality.

Figure 13. Generic Policy Engine Overview

Accountability

Mechanisms must be in place to uniquely identify and authenticate users and to dynamically track their actions within an application or system. Security controls, regardless of type, are not foolproof and to ensure a system of recourse after a security breach a non-circumventable, unalterable, continuous audit mechanism must be in place and operational. A base set of auditable events must exist which can never be disabled. The routines must be non-circumventable ensuring that neither accidental nor intentional modification of the audit system occurs.

The unique identification and authentication of every user or a particular user or system ensures that individuals utilizing the system can be held accountable for their actions. The mechanics behind identification and authentication can vary greatly and as such the core elements should be customizable to suit the threat assessment and the particular architecture. The guarantee of a minimal auditable event list is vital to maximize the trustworthiness of an application

Access Control

Information flow must be controlled so that access to and manipulation of information is restricted to a specific set of individuals utilizing specific tools. As such there must be a mechanism provided to store and manipulate access controls for the various entities that are to be controlled. Each entity contains its own access controls entries to which programmers have access via the Access Controls API. The Access Controls data elements themselves are accessible only to the Access Controls API. The API is the sole method of retrieving pertinent information concerning an entity so that the Security Policy can operate effectively. The Security Policy itself is programmable by the developers though the Security Policy API is fixed. This provides the maximum flexibility: auditable events can be established at the API junction while removing any concern that the developers need remember to insert specific auditable events into the

Security Policy they are writing. The Security Policy is customizable but is restricted to using the Access Controls API to access the Access Controls.

Security Policy

As a general purpose programming language additional security policies can be modeled which enhance, complement, or override the default security policy implemented within each entity. Security policies written in Idyllic can be modified to reflect the requirements of the applications developers thus reflecting the general policies and procedures pertaining to information flow and control for a specific organization. Default security policies, those applicable to all entities, can be provided to simplify security policy maintenance. Furthermore, compatible and cooperative security policies can be written to represent the nuances particular to a given department or section.

Virtual Machine

A virtual machine architecture for Idyllic and the Generic Policy Engine was adopted to maximize portability and simplify the core and primitive elements of the language. Virtual Machines have been shown to be small and efficient [24, 27, 29, 35, 41] and relatively simple to port.

3.1 The Entity

The *entity* is the unified security object against which all security related functions are performed. Both the base security application and any ancillary security relevant applications utilize entities in order to manage and maintain the security attributes of the applications. Figure 14 outlines the base entity within the Generic Policy Engine that represents the actual data stored by the calling application. The entity is subdivided into six basic components: unique identifier, authentication information, audit history, access controls, privileges, and security policy.

Unique Identifier	Defined by the developer, usually reflective of the underlying application or operating system.
Entity Type	The type of entity. There are two distinct types. Those of type <i>group</i> are those which define a group of entities for a particular purpose. Those of type <i>entity</i> are for all other entities. A strict entity can contain multiple groups to which it belongs. A group contains a list of entities forming the particular group. The delineation assists the GPE in quickly traversing associations in order to simplify the security policy's goal of determining whether or not an information flow is to be permitted.
References	What entities reference this entity? This comes in two forms: the first is a true entity which has others referencing it for access controls purposes; the second is when this entity is a group and simply refers to members of the group.
Authentication	String or data required for access (i.e., Password, private key, etc). This can contain more than one authentication attribute, each associated with a mechanism within the security policy.
Security Policy	The security policy for the entity. Each entity can have a different policy or can adopt a general one for the specific machine, domain, network, or institution to which it belongs.
Audit History	
Owner	A history of who "owns" this entity, most recent 1 st , original owner (i.e., creator) last including timestamps.
Last Modified	A history of who last modified this entity, most recent 1 st , original creator of file last including timestamps.
Last Access	A history of who last accessed this entity and by what means, most recent 1 st , original creator of file last including timestamps.
Purge Rate	When events should be purged from the audit log.
Privileges	Historically these have been hierarchical in nature, but more modern security policies require a more flexible alternative.
Groups	To which groups or roles does this entity belong?
Associations	To which associations does this entity belong? Typically these are roles and the like.
Level	What "security level" does this entity have? This is security policy specific in that some require an individual level, others a range, and other none at all. In the cases where it can be within a range, the first element of the list can indicate the currently valid level and subsequent elements can either list the valid range values or provide a minimum and maximum level as per the security policy.
Categories	To which "categories" does this entity belong? This is security policy specific.
Caveats	What "caveats" does this entity have as restrictions? This is security policy specific.
Access Controls	What are the controls associated with this entity? The two most common are Access Control Lists (ACLs) and Roles.
Read	What other entities can request disclosure of information from this entity?
Write	What other entities can request to update the information in this entity?
Execute	What other entities can activate this entity?
Delete	What other entities can request that this entity destroy itself?
Copy	What other entities can request this entity create a copy itself?

Figure 14. Security Attributes of an Entity

In order to map the entities managed by the Generic Policy Engine back to the entities under control of the calling application there must be some unique mechanism in place which can determine absolutely which entity corresponds to which application controlled data item. The calling application will always utilize the unique name when requesting or adjusting information stored and managed by the Generic Policy Engine.

To access some entities it may be necessary to provide appropriate authentication, i.e., entities that hold user authentication information or protect data items that have been password protected by the system or owner. To meet expanding authentication needs and retain compatibility with existing identification and authentication schemes the entity allows for a series of zero or more authentication strings to be stored by the entity and access by the Identification and Authentication API (I&A API).

As changes are performed to the item being protected the entity must track the changes, when and by whom they were performed, and the status of the attempt. This information is maintained by the Generic Policy Engine's audit facility. It allows each entity to maintain their own audit information.

Access controls are, in reality, split into two distinct portions: *privileges* and *access controls*. The former is maintenance information used to track groups, associations, levels, etc. to which this particular entity is a member; included in the list of maintenance information are all those entities which reference this entity. Access controls can be broken out into five standard "controls": read, write, execute, delete, and copy. Any number of these can be utilized, in any fashion required to assist the security policy in performing its duties. It is irrelevant whether the security policy is access control based, capabilities based, role based, or otherwise based. The information required to properly execute such policies is provided within the entity, and especially within the *Privileges* component.

Groups, associations, levels, categories, and caveats provide a mechanism by which like entities can be grouped together. For example, all entities classified *Secret* would have the same Level. Definition of a group, say *Engineering*, could further subdivide the entities classified as *Secret* to those that are *Secret* and available only to someone in *Engineering*. Each definition of a privilege for an entity further restricts access. Similarly, when an entity becomes active, its privileges define the domain to which it belongs and is utilized by the GPE to determine which passive entities can be accessed.

The other privileges provide a further refinement along traditional lines, namely categories and caveats.

The security policy, dependent and written for a particular application, utilizes the access controls and privileges to determine what is considered appropriate information access. A pointer to the security policy is provided within each entity allowing for individual policies per entity *or* for policies to follow the information as it is copied from one host or application to another. In this manner the information understands its policy and the validity of requests made by entities outside its original domain. Thus, applications protected by Generic Policy Engine built security components could move information securely between themselves knowing that not only the base data is transmitted during the flow of information between the applications but that the security information is passed along as well. This can be utilized to ensure that information to which one individual has access remains with that individual, even though the information was moved to a secondary system for whatever reason. Historically, once information left one trusted system with a given security policy to another, the new security policy determined access controls. In this scenario, if the new security policy was not disclosure driven, for example, it might permit accesses to the information that would be forbidden under the original security policy. By having the security policy move with the entity it requires a conscience effort by the owner of the item to modify the security policy to the policy of the new host application along with all that it entails.

The flexibility the entity provides, regardless of overhead, to handle any security policy, individually or in tandem, is unique and revolutionary but does not come without cost. Maintaining entities with numerous security policies poses administrative problems that cannot be adequately addressed in this thesis. Needless to say, the administrative costs would include denial of access to information within an application due to older security policies carried forward by information movement. Procedural solutions could be developed to alleviate most of these access denial situations.

3.2 Idyllic

The base language of the Generic Policy Engine is based on two simple premises:

1. Manipulation of user information must be done efficiently and with minimal overhead.
2. The language must be *provably correct* in order for the GPE to attain the highest levels of trust within the various criteria.

Premise 1 removes most languages from contention, most notably Smalltalk and the other object oriented languages. The class library overhead of OOP languages is unnecessary baggage that in turn impacts on Premise 2.

Premise 2 requires a simple language that is easily understandable. In computer security, the development language for secure components must be *well defined*. To be *well defined* a language must have a formally defined grammar, a stable history (few, well defined and delineated changes over time), and supporting documentation available to the computer industry. Most computer languages (i.e., C/C++, Smalltalk) meet these requirements. However, for a *language* to be provably correct requires that all *aspects* of a language be properly defined and mathematically (or otherwise) provable to function as

advertised. This would include all class libraries, control mechanisms, etc. Classical languages such as C/C++ and Smalltalk have large class libraries that preclude them from easily being proven correct.

The Generic Policy Engine (GPE) is defined within a small functional meta-language, called *Idyllic* (Policy Meta Language) based on the LISP dialect Scheme [43]. The major differences between Scheme and Idyllic are that Idyllic contains no provisions for:

1. lazy evaluation;
2. use of rational or floating point numbers; or
3. for calls with current continuations.

Strings are provided but at a drastically reduced functionality than found in Scheme or LISP. As programs written in Idyllic have no need for I/O, no method of I/O is provided in the production environment, though one is provided via special debugging routines in the debugging environment.

Idyllic's control structures resemble Scheme or LISP but do not share the richness of Common Lisp.

3.2.1 *Why Scheme*

Most functional languages are simple, with a minimum of both control mechanisms and external libraries. LISP-like languages, however, even without external libraries, are fully functional languages whose definition is short and well defined, and whose implementations are sufficiently simple as to ensure ease of trustworthiness. Scheme approaches LISP in a RISC-like manner, creating a reduced instruction set LISP with all the base functionality of LISP [4], with the extensibility, but none of the extra baggage associated with a multi-purpose language. Scheme's designers [43] felt it more important to create a language capable of having its syntax extended rather than providing

everything available in current languages. The reasoning was simple: a simple language would be simpler to learn, quicker to master, and easier to implement.

Idyllic was designed to provide system programmers with an expressive tool capable of elegantly defining security specific functionality. Scheme's ideal, one of a functional language devoid of superfluous functionality but fully capable of providing programmers with the capability of adding necessary additions, is not lost on the goal of the Generic Policy Engine. The Revised Report on Scheme [43] emphasizes Scheme's ability to reduce, via rewrite rules, all of the functionality found in conventional LISPs to a few base primitives and required procedures. This ability goes a long way towards providing its strong base as a provably correct language [29]. By utilizing such a simple language but one with sufficient expressive power to implement the Generic Policy Engine, we gain on two fronts: i) we meet the requirements of the various security criteria; and, ii) we provide an easy to learn system, one which is less likely to be bypassed in favour of direct low-level coding due to complexity issues.

Foregoing the previous argument, one need only examine the various implementations of LISP and Scheme currently available. The most common LISP dialect in use today is Common LISP. The most common dialect of Scheme is R4RS based Scheme. A full implementation of Common LISP requires megabytes of storage, both disk and RAM, and a huge library. Many executables run into the tens of megabytes before anything of value can be accomplished. Scheme, on the other hand, can be implemented in a few thousand bytes and typically operates an order of magnitude faster than LISP implementations. Small, embeddable Schemes exist [24, 34, 35] and are regularly utilized where expressive functional languages are required, but where space is at a premium. Scheme 48 is utilized by NEC [35] to program robots with the resulting program having to fit in less than one megabyte of RAM, including the data area.

Further, designing a language compatible with Scheme provides for other advantages: a known archive of software, a familiar (to many) programming language, proven formalisms, and proven verifiability. Verifiability alone has been approached from differing perspectives resulting in highly provable systems based on Scheme. The work of Peter Lee [40] and MITRE's VLISP [29] to develop a verifiable environment are recent cases in point. VLISP in particular is promising since it showed that a Scheme-like language could be developed into a fully trusted, well-defined programming language with a formal denotational semantics and proofs of correctness and operation. Since all the work by MITRE was performed against an earlier release of Scheme 48 [35], we have opted to use the same Scheme dialect as our base for Idyllic. The advantages of Scheme 48 include a well defined semantics, a verifier from MITRE, a small footprint – image plus VM less than 250k – and a fast run time implementation – 150 times faster than DEC's Scheme->C compiler.

Guttman and his work on VLISP showed the ease with which Scheme and Scheme-like languages could be proven correct and that public domain implementations, most notably Scheme 48, are solid, commercial-quality interpreters with very impressive run-times [29]. Scheme's very nature and its small size were the most useful aspects in formally verifying the correct functioning of the VLISP system. MITRE showed that it was possible to verify a Scheme-like language and its semantics due to the nature of the language itself.

Guttman and Lee both provided similar explanations as to why Scheme provided such an excellent platform for verifiable security environments. They both noted that Scheme has attributes few other modern languages possess, namely:

1. A formal Bachus-Naur grammar;
2. A formal denotational semantics;
3. A simple, elegant design;
4. Small, efficient, fast implementations;
5. A well defined core set of functionality upon which all other functionality is based upon (provable via rewrite rules); and
6. Consistency.

The case for the selection of Scheme as the base for Idyllic becomes stronger still when one considers that it is so close to LISP in syntax. And, although LISP has a wider base of programmers and more programmers have heard of LISP, anyone currently familiar with LISP can program in Scheme quickly and with little difficulty. The minor syntactic and semantic differences are easily overcome. Since both are pure symbolic functional languages, their differences are more at the perceived than real levels.

What follows is a brief description of the various features of Idyllic. It is assumed that the reader is acquainted with at least one dialect of the LISP language.

3.2.2 *Booleans*

The standard Boolean values for true and false are written as `#t` and `#f`, case is unimportant. The empty list, `()`, can be used as false, as per LISP, unlike some implementations of Scheme. In Idyllic only `#f` or `()` are considered to represent false in a conditional expression. `#t` and all non-empty lists are considered to denote true. Note that the symbol `nil` is undefined in Idyllic.

<code>(boolean? obj)</code>	<i>boolean? returns #t if obj is #t, #f, or () and #f otherwise,</i>
<code>(not obj)</code>	<i>Not returns #t if obj is false, and #f otherwise</i>
<code>(or <test_i> ...)</code>	<i>The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a non-false (not #f) value is returned. If all expressions evaluate to a false value, #f is returned</i>
<code>(and <test_i> ...)</code>	<i>The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a false (#f) value is returned. If all values evaluate to non-false values, the last expression's result is returned</i>

3.2.3 Equivalence Predicate

A *predicate* is a procedure that always returns a boolean value (#t or #f). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates found in Scheme or LISP only `equal?` is implemented in Idyllic. This is primarily for reasons of clarity rather than efficiency since `equal?` is usually the least efficient equivalence predicates available.

<code>(equal? obj₁ obj₂)</code>	<i>Equal? returns #t if obj₁ is equivalent to obj₂.</i>
---	---

Note that the advantage of using `equal?` as the only equivalence predicate is that it allows the policy designer to focus on the task at hand rather than on whether or not particular entities are equivalent. `Equal?`'s greatest advantage is that, generally, if two expressions print the same they are equivalent. This is not true of most equivalence predicates available in Scheme and LISP.

3.2.4 Pairs and Lists

All LISP-like languages contain lists, and at a more fundamental level, pairs. A *pair* (sometimes called a *dotted pair*) is a structure with two fields called the *car* and *cdr*⁷ fields. Pairs are created by the procedure `cons`. The *car* and *cdr* fields are accessed by the procedures `car` and `cdr`. The *car* and *cdr* fields are assigned by the procedures `set-car!` and `set-cdr!`.

Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list or a pair whose *cdr* is a list. More precisely, the set of lists is defined as the smallest set *X* such that:

- The empty list is in *X*.
- If *list* is in *X*, then any pair whose *cdr* field contains *list* is also in *X*.

The elements in the *car* fields of successive pairs of a list are the elements of the list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type (it is not a pair); it has no elements and its length is zero.

<code>(pair? obj)</code>	<i>pair?</i> returns #t if <i>obj</i> is a pair, and otherwise #f.
<code>(cons obj₁ obj₂)</code>	Returns a newly allocated pair whose <i>car</i> is <i>obj</i> ₁ and whose <i>cdr</i> is <i>obj</i> ₂ .
<code>(car pair)</code>	Returns the contents of the <i>car</i> field of <i>pair</i> . Note that it is an error to take the <i>car</i> of non-list or of the empty-list, ().

⁷ The original term originates in computer science antiquity. The original LISP interpreter was written on an IBM mainframe which had two primary registers which speeded up processing. These registers were the *address register* and the *decrement register*. When LISP was written *car* was an optimal method of accessing the address register and hence its acronym which stands for *contents of the address register*. Similarly, *cdr* returned the *contents of the decrement register*. These names have survived to this date.

<code>(cdr pair)</code>	<i>Returns the contents of the cdr field of pair. Note that taking the cdr of the empty list returns the empty list⁸.</i>
<code>(set-car! pair obj)</code>	<i>Stores obj in the car field of pair. Pair must be a list. The value returned by set-car! is unspecified. This is a list destruction operation and must be used with great care.</i>
<code>(set-cdr! pair obj)</code>	<i>Stores obj in the cdr field of pair. The value returned by set-cdr! is unspecified. This is a list destruction operation and must be used with great care.</i>
<code>(null? obj)</code>	<i>Returns #t if obj is the empty list, otherwise returns #f.</i>
<code>(list? obj)</code>	<i>Returns #t if obj is a list, otherwise #f. By definition, all lists have finite length and are terminated by the empty list.</i>
<code>(list obj ...)</code>	<i>Returns a new list containing the provided arguments.</i>
<code>(length list)</code>	<i>Returns the length of list. List must be a list.</i>
<code>(append list ...)</code>	<i>Returns a list consisting of the elements of the first list followed by the elements of the other lists. All lists must be lists.</i>
<code>(reverse list)</code>	<i>Returns a newly allocated list consisting of the elements of list in reverse order. List must be a list.</i>
<code>(member obj list)</code>	<i>Returns the first sublist of list whose car is obj. If obj does not occur in list, then #f is returned.</i>
<code>(assoc obj alist)</code>	<i>Returns the first pair in alist whose car field is obj. If no pair in alist has obj as its car, then #f is returned.</i>

3.2.5 Numbers

Idyllic has but one type of number: integers. The reason for not implementing any other form of number is that, historically, operating systems and their components have never required floating point nor rational numbers, let alone complex numbers. To that end, Idyllic contains only integers.

An implementation of Idyllic must support integers throughout the range of numbers that may be used for indices of lists, vectors, and strings or that may result from computing the length of a list, vector, or string. The `length`, `vector-length`, and `string-`

⁸ This is a direct divergence from Scheme which disallows taking the cdr of the empty list. LISP, however, returns the empty list if the cdr of the empty list is attempted. Idyllic opts for the latter for sanity's sake.

length procedures must return an integer. The functions listed below will always return an integer result.

	+	-	*
	div	remainder	modulo ⁹
	max	min	**
(number? obj)	Returns #t if obj is an integer, otherwise returns #f.		
(= x ₁ x ₂)	Returns #t if the arguments are equal, otherwise #f.		
(< x ₁ x ₂)	Returns #t if x ₁ is less than x ₂ , otherwise returns #f.		
(> x ₁ x ₂)	Returns #t if x ₁ is greater than x ₂ , otherwise #f.		
(<= x ₁ x ₂)	Returns #t if x ₁ is less than or equal to x ₂ , otherwise #f.		
(>= x ₁ x ₂)	Returns #t if x ₁ is greater or equal to x ₂ , otherwise #f.		
(odd? n)	Returns #t if n is odd, otherwise #f.		
(even? n)	Returns #t if n is even, otherwise #f.		
(min x ₁ x ₂ ...)	Returns the minimum of the provided values.		
(max x ₁ x ₂ ...)	Returns the maximum of the provided values.		
(abs n)	Returns the absolute value of the provided value.		
(+ x ₁ x ₂)	Returns the sum of the two numbers.		
(- x ₁ x ₂)	Returns the difference of the two numbers.		
(* x ₁ x ₂)	Returns the product of the two numbers.		
(** x ₁ x ₂)	Returns the x ₁ to the power of x ₂ .		
(div x ₁ x ₂)	Returns the quotient of the two numbers.		
(remainder x ₁ x ₂)	Returns the remainder of the quotient.		
(modulo x ₁ x ₂)	Returns the x ₁ modulo x ₂ .		

⁹ *Remainder* and *modulo* differ on negative arguments; *remainder* has the sign of the dividend, while *modulo* has the sign of the divisor.

3.2.6 Control Structures

Idyllic includes but a fraction of Scheme or LISP's control structures. Scheme and LISP allow for numerous looping constructs as well as numerous conditional expressions. Idyllic, on the other hand, has opted for simplicity. If it requires extra lines of code or further thought to create something more easily done in LISP or Scheme, so be it. Programs executing under Idyllic need to be trusted and to be trusted they must be simple. Additional layers of complexity simply cloud the issues of security, complicate formal proofs, and obfuscate correctness.

To that end, Idyllic contains only eight control structures: `apply`, `cond`, `define`, `lambda`, `let`, `map`, `quote`, and `set!`.

<code>(apply fn largs)</code>	<i>Calls the function <code>fn</code> with the elements of the list <code>largs</code>. Returns the value <code>fn</code> typically returns after evaluation. For functions that accept no parameters, <code>largs</code> is the empty list.</i>
<code>(cond <clause> ...)</code>	<p><i>Each <clause> should be of the form</i></p> <p style="text-align: center;"><code>(<test> <expression> ...)</code></p> <p><i>where <test> is any expression. The last <clause> may be an "else clause", which has the form:</i></p> <p style="text-align: center;"><code>(else <expression> ...)</code></p> <p><i>A cond expression is evaluated by evaluating each the <test> of each <clause> until one evaluates to true. The associated list of <expression>s is then evaluated. The final <expression>'s value is returned by the cond. If no <test> expression evaluates to true, then the else clause is evaluated. If no else exists, #f is returned.</i></p>
<code>(define <var> <exp>)</code>	<i>Binds the expression <exp> to the variable <var> at the current level of scope. This is the only method by which a variable can be created into the current environment. Further, definition of a function is performed by having <exp> be a valid λ (lambda) expression. Once bound, a function can be called by simply placing parenthesis around the variable along with any required parameters.</i>
<code>(lambda <args> <body>)</code>	<i><args> is a formal parameter list and <body> is a sequence of one or more expressions. The lambda expression evaluates to a function. This lambda is the only method by which programmable information can be stored within Idyllic. Evaluating the lambda expression along with the appropriate</i>

number of parameters is the method of calling functions in Idyllic. By using define one can store the definitions of the various functions within the current scope.

`(let <bind> <body>)`

The let expression is, in fact, syntactic sugar for a lambda. However, to aid in readability of Idyllic code, the let has been allowed to be the one exception that defines the rule of Idyllic's simplicity. <Bind> is a list of variable/expression pairs evaluated in turn with each expression being evaluated and bound to the provided variable. Once the binding is complete, the <body> is evaluated and the evaluation of the final expression found within the body returned.

`(map fn list)`

List must be a list and fn must be a function taking as many arguments as there are lists. If more than one list is provided, then all must be the same length. Map applies fn element-wise to all elements of the provided lists and returns a list of the results, in order.

`(quote atom)`

Evaluates the provided atom to create a literal constant defined as atom. Quote can be abbreviated as the single quote character `'`.

`(set! <var> <expr>)`

The expression <expr> is evaluated and the resulting value is stored in the location to which the variable <var> is bound. The variable <var> must have been defined within the current scope or within the global scope. A variable is defined within a scope by means of the define structure, above.

3.3 Design

Security is concerned with limiting access between entities to only those authorized by the security policy. Security is usually defined in terms of an entity-based model which defines all entities controlled by the security policy as entities in one of two states: subject, the active state or object, the passive state, see Figure 15. This definition has been shown to be limiting [9, 20, 22] and causes problems when attempting to define security policies across an object-oriented system. The primary issue remains one of state and how the transitions from one state to another are accomplished.

Figure 15. Classical Subject/Object Interaction as per TCSEC

In this thesis we propose the transformation of all subjects and objects into entities. By removing the distinctiveness of subjects and objects the definition of security policies becomes one of restricting flow between peers, rather than between “active” and “passive” entities. An entity's passiveness or activeness is then directly related to whether the entity is actually being acted *upon* or performing the action.

3.3.1 The Interface

In Figure 16 we see an external application, such as an operating system, utilizing a policy written in the GPE. There are two customizable portions to the security policy: the I&A mechanisms and the Security Policy, both indicated in white. Login requests are handled via the standardized GPE API but the login mechanism can be any the developers feel would be most appropriate. This allows for rudimentary login facilities as found on most operating systems today or complex login facilities, such as those based on certificate authorities and X.500 directory services.

Figure 16, presented earlier as Figure 13, presents a stylized structural of the Generic Policy Engine. Each application wishing to utilize the Generic Policy Engine's security features would instantiate their own copy. Each GPE would be protected by the underlying hardware and memory management, ensuring information can't cross application boundaries. It is assumed that the GPE will be utilized in those environment which properly adopt memory management, object reuse, and 32 bit architectures.

Figure 16. Generic Policy Engine Overview

Without the adoption of these elements, the running application cannot be deemed trusted in the evaluation criteria sense of the word. Hence, the Generic Policy Engine would allow for multiple environments by providing for instantiations capable of creating their

own secure environments and namespaces governed by the GPE for a particular domain such as an operating system or a database.

The utility of multiple environments is that various applications can use the Generic Policy Engine to create and manage their security policies and their particular authorization rules. This provides, in the case of an operating system, the ability to provide a *fully* functional security API for applications to use courtesy of the operating system and one that is as powerful as the native operating system's security system. Since all security is handled by the GPE, it can resist tampering and ensure logs are properly maintained.

Once authentication to the security policy has been achieved, all information access requests made to the Policy API are either boolean or string in value. Boolean values are returned when a request is made for specific access to a given controlled data item, such as a file. Strings are returned when the request must return other than true or false, such as a cryptographic key, the true location of a file, current access controls for a particular controlled entity, etc.

3.3.2 Identification and Authentication

It is impossible to have a security system without a mechanism of identification and authentication. Historically, identification and authentication (I&A) referred to the login/password pair which greeted users prior to the system allowing the user to begin processing. As time progressed, so has the I&A mechanisms until today we have numerous technologies ranging from login/password pairs to one-time passwords to crypto-tokens.

For the Generic Policy Engine to be generic from an I&A point of view it must be sufficiently flexible to allow one or more varied identification and authentication mechanisms. To ensure that programming for the I&A mechanisms chosen remains as

simple as possible, a fixed API is required guaranteeing programmers are met with a uniform interface regardless of underlying mechanism. To that end, the Generic Policy Engine must provide a standardized I&A API which interfaces to application specific identification and authentication mechanisms.

3.3.3 *Audit*

Further security is added by providing full audit trails that are customizable by the systems programmers. Since the security policy is available for perusal by the designers, the audit requirements of a particular application can be customized as required. As little or as much audit data can be written to various logs as required. The particular information written to the logs is also under the full control of the designers and no limits are placed on the complexities of the source defining how the audit records are created. This provides further flexibility in granting the security policy the option of calling helper functions to create detailed audit records in some instances while in others the records could be less detailed prior to pushing them out through the standardized audit mechanisms provided by Idyllic and the Generic Policy Engine. There exists, at all times, a lowest common denominator of auditable events that cannot be turned off. These are hard coded into the actual APIs ensuring that the system programmers cannot accidentally circumvent the audit system.

3.3.4 *Access Control*

There are many forms of access controls. The primary form are known as confidentiality access controls, controls which contain to whom information can be released. Conversely, the control of who can manipulate information stored by an application are known as integrity access controls and operate along similar lines.

Access controls are further divided into the mechanism employed. The classic method is the use of access control lists that place the control information on each entity with

explicit user names as to who can and cannot access the information. A modification of this form is role-based access controls whereby users are associated with roles and it is the role by which each entity determines whether or not an individual can have access. Other access control mechanisms operate similarly.

The Generic Policy Engine must be flexible enough to handle any form of access control, regardless of whether the controlling action is performed by the active entity, passive entity, or some combination thereof. The entity defines the information pertinent to the decision making of the security policy and contains sufficient information to model any known security policy. Since each entity controlled by the Generic Policy Engine is actually defined *by* an entity, the security policy defined is not restricted to a single form of access control. Thus, the application developer can define security policies that utilize information stored in an entity, regardless of state (active or passive) and determine via appropriate combination the access permissions.

For example, if the access control model is role-based then passive entities only indicate the type of access associated with a given role. The user entities actually define the roles to which they belong. Thus, if *User1*'s only role is *clerk* and *Entity1* only allows a *clerk* to read its contents, then *User1* will be granted read permission. If *User1* is removed from the system, *Entity1* does not need to be modified since it has no direct reference to *User1*. The same cannot be said for access control lists that reference the actual users in each entity's access list.

3.3.5 Object Reuse

The entities under the control of the Generic Policy Engine are not the true data items being controlled but rather encapsulators defining the security attributes of the data. The Generic Policy Engine, or more accurately Idyllic, automatically garbage collects deallocated memory in order to ensure that information between entities is not accidentally shared. The calling application must perform object reuse properly in order

to ensure the data being protected isn't accidentally compromised. The Generic Policy Engine does, however, provide an object reuse call that the application programmer utilizes to inform the Generic Policy Engine a specific entity is no longer required. This call is part of the Generic Policy Engine's API defined in section 3.5.8, below.

3.3.6 *Trusted Path*

The Generic Policy Engine has no mechanism to ensure that a trusted path has been set up between the application and the end user. This physical mechanism must be provided by the application whenever the user issues a predefined key sequence to the application. The security policy for the application can be coded such that it requires specific information prior to performing specific actions, such as information downgrades. This then becomes an issue addressed by the programmable security policy.

3.4 Implementation

This section takes the design information presented above and provides actual code for the various pieces presented, fleshing out the entity data structure, its associated functions, and the resulting class-like structure.

3.4.1 *The Active Data Structure*

In most cases, data structures are passive entities against which a system or application applies procedures or functions. In the Generic Policy Engine a single, state-aware data structure represents all the actual items within a system or application being secured, be they passive, such as files or active, such as processes. The entity can be in one of two states: active or passive. The first state is associated with information processing and typically operates against passive entities. To any active entity all other entities in a system appear passive. This allows for the manipulation and access of entities by entities to be generalized.

```

Entity      = STRUCTURE
    UniqueIdentifier: string
    EntityType:      Atom
    References:      LIST of Entity
    Authentication:  string
    SecurityPolicy:  Lambda, Default = NOACCESS

    /* Audit Logs

    Owner:           LIST of Entity
    LastModified:    LIST of (Entity, Timestamp)
    LastAccess:      LIST of (Entity, Timestamp)
    PurgeRate:       Lambda, Default = NEVER

    /* Privileges

    Groups:          LIST of Entity
    Associations:     LIST of Entity
    Level:           LIST of Entity
    Categories:      LIST of Entity
    Caveat:          LIST of Entity

    /* Access Controls

    Read:            PAIR (Allow: LIST of Entity,
                          Deny:  LIST of Entity)
    Write:           PAIR (Allow: LIST of Entity,
                          Deny:  LIST of Entity)
    Execute:         PAIR (Allow: LIST of Entity,
                          Deny:  LIST of Entity)
    Delete:           PAIR (Allow: LIST of Entity,
                          Deny:  LIST of Entity)
    Copy:            PAIR (Allow: LIST of Entity,
                          Deny:  LIST of Entity)

    ENDSTRUCTURE

```

Figure 17. Pseudo-code Data Structure for GPE Entity

Further, it is the entity, defined using pseudo-code in Figure 17, itself that understands how it is to be manipulated or accessed and can control the type of access via the security policy embedded within it. This is accomplished by storing the security policy for each entity within the entity. This allows the security policy to travel with the entity, regardless of environment. Thus a user requesting access to a passive entity would have to be allowed access by the security policy of the entity itself, even though this policy may be a general one applicable to and shared by all entities in the system. By providing each entity with the ability to carry its own security policy the Generic Policy Engine provides a

mechanism by which the information can be transmitted to another GPE controlled system, retaining the original access controls and security policy.

Idyllic implicitly manages all its data structures as symbolic expressions. The entity is by definition a symbolic expression and therefore properly handled by Idyllic. In order to differentiate between entities, each entity is uniquely identified so as to be accessible locally, internally to Idyllic, and externally to the calling application. Each entity has a current state, which indicates whether it refers to a user, process, or data. The data remains private with access permitted only via an exported API.

In a manner of speaking, an entity can be viewed as a database entry where the database management system is Idyllic and the database language is the GPE's APIs. The entity is managed as are all symbolic expressions within Idyllic. Accessing and manipulating entities via the APIs provides the security functionality associated with audit, identification and authentication (I&A), access controls, privileges, and security policy. The APIs can be subdivided into:

- General Entity API
- I&A API
- Audit API
- Access Controls API
- Privileges API
- Security Policy API

The API is used in order that the entity be useful outside the bounds of Idyllic. The entity is stored as a series of symbolic expressions and each is uniquely identified and mapable to an external calling application. This one-to-one relationship between the external representation of the entity and the GPE representation of the entity is vital to the security and integrity of the system. As each entity is manipulated it is fetched and stored in the access controls database. Each entity is *identical* in look and can be utilized for any purpose. This lack of typing is crucial to ensure that the GPE be sufficiently flexible in

implementing any form of security policy utilizing its framework. The structure of the entity is provided in Figure 18.

```
(define Entity
  (UniqueIdentifier      (empty-list))
  (EntityType            [Entity | Group | Association |
                          Level | Categories | Caveats])
  (References            (empty-list))
  (Authentication        (empty-list))
  (SecurityPolicy        defaultSecurityPolicy)

  ;* Audit Logs

  (Owner                 (empty-list))
  (LastModified           (empty-list))
  (LastAccessed           (empty-list))
  (PurgeRate              (lambda () #f))

  ;* Privileges

  (Groups                 (empty-list))
  (Associations            (empty-list))
  (Level                  (empty-list))
  (Categories              (empty-list))
  (Caveat                  (empty-list))

  ;* Access Controls

  (Read                   (list '(allow ,(empty-list))
                                '(deny ,(empty-list))))
  (Write                   (list '(allow ,(empty-list))
                                '(deny ,(empty-list))))
  (Execute                 (list '(allow ,(empty-list))
                                '(deny ,(empty-list))))
  (Delete                  (list '(allow ,(empty-list))
                                '(deny ,(empty-list))))
  (Copy                   (list '(allow ,(empty-list))
                                '(deny ,(empty-list))))
)
```

Figure 18. Idyllic Coded Data Structure for Entity

Figure 19 defines the “defaultSecurityPolicy”. It is an explicit denial security policy. In other words, any action taken while this security policy was in force would be summarily denied. It must be changed, a conscious decision to override the default with a custom built security policy. It is in the interest of the application developer to utilize an appropriate and secure security policy.


```

(define (defaultSecurityPolicy . parameters)
  ; *
  ; * The default security policy declines every request for
  ; * information. This follows the standard edict of computer
  ; * security: if not explicitly allowed, it is denied. This
  ; * forces the developer to create a security policy. If the
  ; * developer opts to replace the "#f" with a "#t", it is a
  ; * conscience effort to forego creating a valid security
  ; * policy.
  ; *
  #f)

```

Figure 19. The Default Security Policy for the GPE

The choice of a monolithic entity encumbers the software by making it less modular. According to the current object-oriented programming style, a structurally cleaner and aesthetically more pleasing solution would be to have the various audit logs refer to an audit log, which in turn would contain the methods used to manipulate the logs. High-end security (e.g., the CTCPEC's T-5 and T-6 or the Orange Book's B3 and A1) require a tight coupling of all security relevant data within the GPE. In order to provide the tightest coupling, the GPE couples all the elements of the entity – the audit logs, the I&A, the access controls, privileges, and the security policy – directly to the data being controlled. This, in turn, results in all the security data elements being controlled and stored within a well-defined, controlled area, easily placed under the auspices of a memory manager or other well-defined hardware device.

In order to utilize a true object-oriented approach, the linkages between entities and the notion of what constitutes an entity would have to be written into the underpinnings of *Idyllic*. Tracking and understanding the couplings for a given entity would require extensive syntactic extensions at the primitive level removing much of the simplicity and elegance found in *Idyllic*. This increase in complexity would not provide any additional security or trustworthiness to the GPE. Understanding the relevance of the links while utilizing memory management techniques to track and isolate the various memory elements that constitute an entity would be creating an artifice of memory management for the sole purpose of adhering to an object-oriented paradigm when one is not required.

The use of a monolithic entity has been chosen to simplify and address the issues at hand, namely the creation of a security solution capable of generically describing and implementing security policies for any situation.

One question that does arise is whether a programmer has the flexibility to create additional audit logs or I&A mechanisms. Should the need arise, additional security logs or I&A mechanisms can be provided by utilizing existing mechanisms within the entity while ignoring other present capabilities. Capabilities not utilized can be viewed as being dormant, but available should the programmer wish to utilize them at a later time.

For each application requiring the functionality of the GPE a copy of the GPE is instantiated with its own memory area and entities. This adheres to the data separation requirements of many security criteria [22, 54] while simplifying the implementation. This also provides a clean and elegant solution to providing unique security policies to any application within a system requiring security functionality.

Figure 20 presents the entity with the elements expanded to include stylized contents within their association lists. Each entity has a unique identifier *<uid>* to uniquely identify each entity to the calling application, to the GPE, but also to allow for full cross-referencing. The cross-referencing is used to quickly remove all references to the entity should it ever be required. Hence, each time one entity refers to another, for whatever reason, the referenced entity is updated. Some fields contain lists that refer to various data instances such as time or entity identifiers (uid). For example, *owner* is a list of all the individuals who have ever owned a particular entity, with the first owner listed being the current owner and the last owner being the original owner, or creator. Similarly, *LastModified* lists in order, from most recent to least recent, which entities performed modifications to the entity.

```

(define Entity
  (UniqueIdentifier      <uid>)
  (EntityType            [Entity | Group | Association |
                          Level | Categories | Caveats])
  (References            (<uid> ...))
  (Authentication        (<identifier> ... ))
  (SecurityPolicy        defaultSecurityPolicy)

  ;* Audit Logs

  (Owner                  (<uid> ...))
  (LastModified            ((<timestamp> <uid>) ...))
  (LastAccess              ((<timestamp> <uid>) ...))
  (PurgeRate               (lambda () #f))

  ;* Privileges

  (Groups                  (<uid> ...))
  (Associations             (<uid> ...))
  (Level                   (<currentLevel> <min> <max>))
  (Categories               (<currentCategories> <fullSet>))
  (Caveats                  (<currentCaveat> <fullSet>))

  ;* Access Controls

  (Read                    ((allow (<uid> ...))
                           (deny  (<uid> ...))))
  (Write                    ((allow (<uid> ...))
                           (deny  (<uid> ...))))
  (Execute                  ((allow (<uid> ...))
                           (deny  (<uid> ...))))
  (Delete                   ((allow (<uid> ...))
                           (deny  (<uid> ...))))
  (Copy                     ((allow (<uid> ...))
                           (deny  (<uid> ...))))
)

```

Figure 20. Data Structure for Entity with Expanded Association Lists

Although it would be efficient in design terms to define the various audit logs as separate definitions pointing towards an *AuditLog*, it is more beneficial to define the logs within each entity so that the audit information can be properly protected. Access to the audit logs, elemental to the sanity of the entity and as a mechanism to provide additional warranties that the security has not been breached, requires access to the entity. Replication of I&A information and the possibility for error is removed, albeit at the expense of some processing overhead during audit log review.

Of particular interest in Figure 20 is the use and reuse of the unique identifiers <uid>. Everything within the GPE is an entity, and this includes security levels, categories, groups, associations, and caveats. They are utilized and referenced by the security policy, which places specific information within each. Thus, *Level* would contain pointers to such entities as *Secret*, *Unclassified*, and *Top Secret*. The security policy would be able to request which the *Level* for a particular entity, compare it to other entities, and determine the validity of information access requests. Similarly, a group of entities can be placed into a particular role via the *Associations* tag. The <uid> for the entity could be *Programmer* and reference the entities which are programmers. And again, *Categories* can be used to define category subsets for use by the security policy. Thus a banking security policy could define valid categories as *Teller*, *Bank Manager*, *Branch Manager*, *Loan Manager*, *Clerk*, and *Financial Analyst* and compartmentalize information appropriately. This cross-referencing could be graphically illustrated to visually create representations of how information, one way or another, is related. This could greatly assist the security officer in determining whether there are any information flows that should not exist, such as data path from a higher hierarchical level to a lower one.

By providing a unified interpretation for *Entity* the GPE is generic. It is the interpretation of the information stored within the various elements of the *Entity* by the security policy that determines the form of the security policy, be it role based or access control based, or otherwise. Generic use of the entity generalizes security to its constituent components.

3.4.2 *Entity Based*

In any secure system it is vital that the security information remain protected at all times and that the mechanisms utilized to access the information remain immutable. Object oriented technologies provide a partial solution by formalizing the notion of data hiding with both the data and the procedures to manipulate the data stored within an object defined by a class. This notion can be extended to assist us in creating a viable storage mechanism for the Generic Policy Engine. In the previous section we described the basic

building block of the GPE, the Entity. Each entity corresponds to an item on the application side that the application programmer wishes to secure. The entity is the manifestation of the security attributes of the data in the application.

The combination of the data found within the entity and the procedures to manipulate the entity constitute the entity-based form of the GPE. This works well for static procedures which remain constant. The GPE must allow the flexibility to insert custom portions that reflect the unique identification and authentication mechanisms and the unique security policies of various organizations. At the same time, it must be able to put forward a well defined interface and functionality that would meet international security criteria. The solution is to utilize a standard API for each of the procedures while allowing specific procedures, such as the Security Policy and Identification and Authentication routines, to have replaceable internals.

The following sections present the various procedures and their respective APIs. The final section presents the entire entity-based “class” for GPE-Entity that contains the data described above and all of the APIs and associated functions for manipulating the data within the confines of good security practice.

3.4.3 *The GPE Entity Definition*

Creating a new entity requires no more than a call to *define-entity*. This special form accepts only a single parameter: the entity’s name. It must be unique. *Define-entity* will ensure that the identifier provided is unique and if not, will issue an error. Figure 21 provides a stylized outline of what *define-entity* creates. It is these structures that are maintained and manipulated by the GPE. They encapsulate the security of the items referred to by the calling application. Idyllic and the GPE maintain all the created entities in a persistent state. Each entity is a self-contained security module capable of determining whether or not another entity is granted access and the type of access allowed. Access to the entity is available only through the defined APIs.

```

(define-entity <uniqueID>)

creates

<uniqueID> <=
  (variables (<var> <value>)
    :
    (<var> <value>))

  (private-methods (message . args)
    ((message) <body>)
    :
    ((message) <body>))

  (public-methods (message . args)
    ((message) <body>)
    :
    ((message) <body>)))

```

Figure 21. General Structure of an Entity

Figure 22 provides a completed entity structure for a generic entity, slightly stylized for readability. As will be noted, many portions of the object-oriented paradigm are utilized in the design and implementation of the entity. However, it cannot truly be called object oriented as the only elements of object oriented computing utilized in the GPE are garbage collection (inherent in *Idyllic*), polymorphism, and encapsulation. Inheritance is notably absent. This is the primary reason why the GPE is referred to as entity-based.

```

(define-entity GenericEntity)

creates

GenericEntity <=

  ((variables (UniqueIdentifier GenericEntity)
    (EntityType Entity)
    (References ())
    (Authentication (Active <Expiry> NoPassword))
    (SecurityPolicy defaultSecurityPolicy))

    ; * Audit Logs

    (Owner (<creator>))
    (LastModified ((<timestamp> <uid>)))
    (LastAccess ((<timestamp> <uid>)))
    (PurgeRate (lambda () #f))

    ; * Privileges

```

```

        (Groups          ())
        (Associations    ())
        (Level           (<currentLevel> <min> <max>))
        (Categories      (<currentCategory> <fullSet>))
        (Caveats         (<currentCaveat> <fullSet>))

; * Access Controls

        (Read            ((allow (GenericEntity
                                <creator>))
                           (deny  ())))

        (Write           ((allow (GenericEntity
                                <creator>))
                           (deny  ())))

        (Execute         ((allow ())
                           (deny  ())))

        (Delete          ((allow (GenericEntity
                                <creator>))
                           (deny  ())))

        (Copy            ((allow (GenericEntity
                                <creator>))
                           (deny  ())))

    ) ; * end variables

; *
; * Typical housekeeping functions (BetterHomes&Gardens)
; *

    (private-methods (message . args)
      ((getReferences)
       <code>))

; *
; * The APIs, though their "C-like" form, look nothing
; * like what we have here. The APIs in reality are
; * syntactic sugar but use these calls. Pure GPE coding
; * would be too cumbersome otherwise.
; *

    (public-methods (message . args)
      ((get)
       <code>
      )
      ((put)
       <code>
      )
      ((remove)
       <code>
      )
    ) ; * end public-methods
  ) ; * end entity definition

```

Figure 22. Generic Definition of an Entity

UniqueIdentifiers refer to the *<uid>*s found within the GPE entity. This provides closure between the various classes, linking them together explicitly in a large lattice. Code fragments have been removed for clarity and brevity. The full listings can be viewed in the appendices. *<creator>* is the creator of the new entity. Each entity is created by another entity, which is its creator. All entities can trace their heritage back to the original entity, the system entity, similar to class Object in Smalltalk. The other parenthesized elements (*< ... >*) are self-explanatory.

3.4.4 *Trusted Computing Base*

A Trusted Computing Base (TCB) is the term that defines a boundary within which all controlled entities reside. In trusted systems, developers typically define which entities are under the control of the reference monitor and which are not. The defined subset of entities are considered to be *inside* the TCB boundary while the remainder are considered to reside *outside* the TCB boundary. The application using the GPE must provide the additional security features of memory management and non-circumventability of the security policy.

By default the GPE provides a security policy. This security policy must be invoked by the calling application in order to properly function. There is no mechanism by which the GPE can guarantee the proper invocation, but by simplifying the mechanics behind the actual call, we can ensure that overhead and complexity are removed as viable answers to not utilizing the security procedures of the GPE.

Assuming the security policy is always invoked by the application, the application must be able to ensure the information being protected, say files within an operating system, cannot be accessed by standard operating system calls, such as low level disk reads. Although this latter form of access can be viewed as circumventing the security policy, many times this form of access is ignored especially when considering direct memory

accesses. Modern memory management assists in preventing cross boundary access and alleviating many attempts at direct manipulation of memory.

3.4.5 *Reference Monitor*

In order for the GPE to be considered “trusted” it must meet the requirements which define a *reference monitor*:

- tamperproof;
- always be invoked; and
- small enough to be subjected to analysis and tests to ensure its correctness.

Does the GPE satisfy these requirements?

To be tamperproof the GPE must reside within a protected memory space. Modern computer hardware provides such mechanisms which are difficult to circumvent and meet the requirement of non-circumventability. Invocation requires that the application being protected *always* invoke the GPE for every transaction. This is something that falls to the application developer to address and if security is paramount, then invocation will occur. The final requirement is met by ensuring that both the GPE, Idyllic, and any security policies written in Idyllic are simple enough to be subjected to analysis and shown correct. As defined in the next section, policies that typically took thousands of lines in classical security policies can be defined in less than a hundred lines using the GPE.

Inheriting Trust

One of the most difficult areas in computer security is ensuring that the system being developed is trustworthy. By using reference monitors and trusted computing bases, it is possible to increase the amount of trust that can be placed in a specific trusted product. However, for each secure product developed, all aspects must be examined by an

approved evaluation authority. This is a time consuming process. One of the main goals of the GPE is to reduce the amount of time required to evaluate a trusted product.

The Matter of Persistence

The controlled entities must be readily available between invocations. At present Idyllic stores all of the entities in its address space, readily available to the GPE. On true, large scale applications a caching scheme must exist capable of ensuring entities are fetched and stored to disk depending on usage and when modified. Modifications to an entity must be written to disk immediately to minimize the likelihood security information would be lost for whatever reason. The caching scheme would ensure that the most used entities remain in RAM, those less often used would be removed from memory after Idyllic guarantees that the information has been saved properly.

The simplest method of providing persistence is for the operating system to always keep the GPE and its associated entities in active memory. This would also provide for the quickest reaction time but may not be expedient when the number of entities controlled becomes large, as inevitably will be the case. Regardless of system used to keep the entities readily available, it must ensure the isolation, availability, and correctness of all GPE data. Caching schemes are sufficiently well understood that one could be adopted for use by Idyllic to guarantee the integrity of its dynamic namespace.

Upon system shutdown, the GPE would have to be called to ensure orderly shut down of the security policy and proper storage of all databases. In a production version of the GPE a shutdown function would be used to initiate security shutdown. Similarly, when an application shuts down, it must properly terminate any copies of the GPE it instantiated.

3.5 The Entity Application Programming Interfaces

Two versions of the API exist: the internal, LISP-like API and the external, C-like API. The C-like API provides standard procedural prototypes for using the capabilities of the GPE. These C-like calls are converted into the internal LISP-like API. Provided in subsequent sections are the C-like APIs. The full LISP-like API is provided as part of the full definition of the Entity found in Section 3.4.3.

Of note is that all LISP-like API calls are found within the entity as the exported public functions available to manipulate the data contained within. This gives the entity an object-based feel, retains strong encapsulation, and provides a single, cohesive unit to refer to the protected item in the calling application.

3.5.1 *Instantiating a Copy of the GPE*

The GPE's purpose is to provide security to a calling application. Each application requests its own instantiation of the GPE. Each instantiation provides a fully functional copy of Idyllic and the GPE APIs. There are four function calls, defined in Figure 23, divide into two calls for opening a namespace of GPE entities for manipulation and a third to close the namespace. In order for the GPE to operate, a namespace must be opened and available to the calling application.

gpeID = gpeCreate ()

Returns a reference to a new namespace. The application is expected to use gpeID whenever it requires access to the security functionality. If gpeCreate () could not create a new namespace, 0 is returned. gpeID is a long unsigned integer.

Once opened, all initialization to the global status of the GPE must be performed.

gpeOpen (gpeID)

The application requests a previously opened GPE namespace to be opened by calling the function with gpeID. If the namespace exists, #T otherwise #F.

Once opened, all initialization to the global status of the GPE must be performed.

gpeStart ()

All initialization and global sets must occur prior to the

	<i>gpeStart ()</i> command being issued. This ensures that certain attributes are frozen for the running duration of the GPE, such as audit level.
<i>gpeClose</i> (gpeID)	Close the namespace associated with gpeID. #T on success, #F on failure.

Figure 23. Instantiating a copy of the GPE

3.5.2 The Unique Identifier

Every entity has a unique identifier provided by the calling application. This unique identifier (*uid*) uniquely identifies each entity managed by the GPE and provides a consistent mapping from the GPE entities back to the controlled items of the calling application. This simple, one-to-one mapping enhances the reliability and security of the GPE and ensures that identifier translation routines are not required thereby removing a source of possible error.

3.5.3 The Entity API

The GPE Entity API refers to what is normally called *class methods* in object-oriented programming. The functions defined in this API, see Figure 24, allow the programmer to manipulate the entity namespace in a coarse manner; strictly at the entity level.

<i>gpeEntity</i> (uid create)	Creates a new entity within the namespace with the unique identifier provided. All information for the entity must be filled in piecemeal. Returns #T on successful creation, #F otherwise.
<i>gpeEntity</i> (uid remove)	Removes the entity referred to as uid. Returns #T on success, #F otherwise.
<i>gpeEntity</i> (uid SP securityPolicy)	Sets the security policy for the entity referred to as uid to securityPolicy. SecurityPolicy must be a valid lambda expression. Returns #T on success, #F otherwise.
<i>gpeEntity</i> (uid SP DEFAULT)	Sets the security policy back to the GPE default. Returns #T on success, #F otherwise.

Figure 24. The Entity API

3.5.4 Identification and Authentication API

The GPE Identification and Authentication (I&A) API provides access to the I&A data of an entity. These calls allow for the creation of entities and the definition of unique identification and authentication of entities, regardless of whether they refer to a user or data object externally. The calling application need not track which users are valid nor which privileges they have. The GPE API provides system calls that can authenticate individuals. The GPE API provides additional system calls to add, remove, and update information concerning users of the application. The I&A database is maintained and protected by the GPE within each unique entity¹⁰. Figure 25 provides a summary of the Identification and Authentication API.

<i>gpeAuth</i> (uid Add authStr)	<i>Adds uid, plus its authentication string (authStr) to the entity referred to by uid. Returns #T on success, #F otherwise.</i>
<i>gpeAuth</i> (uid Remove)	<i>Removes uid (i.e., the entity it refers to). Returns #T on success, #F otherwise.</i>
<i>gpeAuth</i> (uid equal? authStr)	<i>Is the uid and authentication string (authStr) correct? #T if correct, #F otherwise.</i>
<i>gpeAuth</i> (uid Expires date)	<i>Sets the expiry date for uid. #T on success, #F on failure</i>
<i>gpeAuth</i> (uid Expires?)	<i>Returns #F if the uid's password never expires, the date of expiry otherwise.</i>
<i>gpeAuth</i> (uid SetPaswd old new)	<i>Sets the password to the new one provided the old password matches the existing one. Returns #T on success, #F on failure.</i>
<i>gpeAuth</i> (uid SetPassLen newlen)	<i>Sets the password length to the new one provided so long as it is between the predefined¹¹ min and max password lengths. Returns #T on success, #F on failure.</i>
<i>gpeAuth</i> (uid deactivate)	<i>Deactivate the uid. #T on success, #F otherwise.</i>
<i>gpeAuth</i> (uid activate)	<i>(Re)activates the uid. Returns #T on success, otherwise #F.</i>

Figure 25. Identification and Authentication API

¹⁰ Note that UID refers to the unique identifier associated with each entity stored and managed by the Generic Policy Engine. In order to properly use the GPE, calling functions must at all times refer to the entities being manipulated by their UIDs. These UIDs are provided by the calling program.

3.5.5 Audit API

The Generic Policy Engine provides a built in audit capability capable of auditing every request by the calling application for mediation between a requesting entity and the targeted entity. Each entity, when accessed, logs all requests, the time of each request, and the status of the request to their internal logs. Each entity has a default audit level. These audit levels can be adjusted so more or less information is gathered. Therefore, to collect less information the audit level is lowered; to gather more, the audit level is raised. Entity level auditing cannot be completely shut off since a base number of auditable events *must* be tracked for security reasons, such as entity create, access, delete, etc. These typically correspond to system level actions such as file open, file close, file delete, add user, remove user, etc. Logs are lists with each entry stored as a sublist. Each entry contains the following information: requesting entity, time of request, action requested, and return status of request. This information is available, typically to the security officer and the audit tools, via the Audit API.

There exist four logs associated with each entity: *default log*, *owner log*, *last modified log*, and *last accessed log*. The *default log* is provided to allow calling application to insert specific information on events the application feels are security relevant. The *default log* can be used to create general logs by creating an entity whose entire purpose is restricted to logging information on behalf of the application. The *owner log* tracks who the owner of the entity is from the entity's inception through to its final destruction. The *last modified log* tracks all attempts at modification to a particular entity. And, the *last accessed log* tracks all accesses made to a particular entity.

¹¹ Minimum and maximum password lengths are set at initialization time for the calling application and are not changeable thereafter. Any change to the required password length must fall within the predefined boundaries. Any changes won't take effect until the system or application is restarted.

<i>MinimalAuditLevel</i>	<i>The only auditing done is the bare minimum, namely recording entity creates and destroys.</i>
<i>AuditLevel1</i>	<i>Same as MinimalAudit but with the inclusion of open entity and close entity requests.</i>
<i>AuditLevel2</i>	<i>Same as Level1 but with the inclusion of modifications (write) to an entity.</i>
<i>AuditLevel3</i>	<i>Same as Level2 but with the inclusion of access (read) to an entity.</i>
<i>AuditLevel4</i>	<i>Same as Level3 but with the inclusion of copies so as to track if information is being moved from user to user via copies rather than standard reads.</i>
<i>MaximumAuditLevel</i>	<i>Same as Level4.</i>
<i>DefaultAuditLevel</i>	<i>Set to one of the above audit levels.</i>

Figure 26. Defined Audit levels In the GPE

For different applications, different levels¹² of audit are required. Some require no more than the rudimentary audit granularity of when an entity was created and when it is modified. Others require a finer granularity capable of indicating whenever an entity has been accessed, or attempted to be accessed, by another entity. As the granularity becomes finer, the amount of information stored increases. This finer in granularity provides a fuller picture of what is occurring security-wise, but requires a much larger storage allocation or more frequent examination of the logs. The GPE defines a specific set of audit levels, as defined in Figure 26. These are adjustable prior to the issuance of the *gpeStart ()* command. Once the GPE starts, these global audit values are immutable.

¹² Levels were utilized to closely correlate to the notion of audit found in modern evaluation Criteria which require distinct audit levels, each finer than the previous.

<i>gpeLog</i> (GetLevel)	<i>Return the currently set level of audit.</i>
<i>gpeLog</i> (SetLevel <level>)	<i>The GPE provides predefined levels of granularity for audit. <level> must correspond to one of the predefined audit levels. Returns #T on success, #F on failure.</i>
<i>gpeLog</i> (setOptions ...)	<i>Sets the option list to the list provided (future).</i>
<i>gpeLog</i> (SetGMT)	<i>Sets the time to GMT, default for Time is local time.</i>
<i>gpeLog</i> (SetLocal)	<i>Sets the time to local time, this is the default.</i>
<i>gpeLog</i> (Time)	<i>Returns the current setting, either Local or GMT.</i>

Figure 27. Audit Log API (Global)

Manipulating the fixed elements is provided by a set of global audit functions, defined in Figure 27. These are the functions which must be called prior to *gpeStart* ().

As time progresses in any security system, logs grow increasingly larger until they occupy all available space. There must be some mechanism to limit their size. In the GPE each entity has a lambda expression which defines the purge rate. This function determines when the log is to be purged, be it by age or by size. If by size, the log is truncated by removing the older data and retaining the newer information. If logs are never to be removed, then the purge date must be set as *never*. This ensures that the GPE never removes the logs but will, if the logs become full, halt the GPE.

Manipulating the logs requires an entirely different set of functions. Typically logs can be accessed in one of two ways: read and append. Many modern logs do not require the selective editing of the logs but rather only the wholesale purge, which is logged in the new log created after the existing log is closed for purging. This ensures a trace exists of all actions throughout the life of the system. The destructive functions are typically restricted to the *Security Officer* user.

Figure 28 defines the common log access functions. The two system calls provided allow access to the four types of log, *defaultLog*, *Owner*, *LastModified*, and *LastAccessed*; *<log>* refers to one of these.

<i>gpeLog</i> (uid put <i><log></i> data)	<i>Appends data to <log> for the entity defined by uid.</i>
<i>gpeLog</i> (uid get <i><log></i> expr)	<i>Returns all entries for the entity uid which match expr. expr can be a wildcard, which returns the entire log.</i>

Figure 28. Audit Log API (Entity Specific)

3.5.6 Privileges API

Privileges in the GPE retain grouping information. There are two distinct types of group: hierarchical and non-hierarchical. Both are defined and accessible to security policies within the GPE framework and both are accessed using the calls defined in Figure 29.

All relevant grouping information is stored within the entity. Groups, associations, categories, and caveats are all non-hierarchical grouping mechanisms. They follow the common mechanisms found in many security policies which provide for the ability to group users or information in specific ways. For example, UNIX allows users to be grouped so as to provide specific groups with certain functionality. The military groups information via categories and caveats in order to limit who within a specific organization, regardless of clearance, can actually see the information. For example, information labeled *Secret NATO* would be visible only to those individuals holding a Secret clearance with the NATO category. Categories and caveats are often called *compartments*.

Hierarchical groupings are typically found in the military where the hierarchy is directly related to the classification system used to store information. The most common classifications are *Unclassified*, *Confidential*, *Secret*, and *Top Secret*. As a hierarchy, each is more restrictive than the former. Thus, Secret information is extremely sensitive but

less so than Top Secret information. Typically, individuals with access to a higher level in the hierarchy also have access to information at lower levels in the hierarchy. Manipulation of the hierarchical mechanisms is performed in exactly the same manner as for non-hierarchical information.

<i>gpePriv</i> (uid put <priv> uid2)	Adds uid2 to the <priv> list for uid. Returns #T on success, #F otherwise.
<i>gpePriv</i> (uid remove <priv> uid2)	Removes uid2 from uid's <priv> list. Returns #T on success, #F otherwise.
<i>gpePriv</i> (uid get <priv>)	Returns the complete <priv> list for the given uid.

Figure 29. Privileges API

Privileges usually form the most secure mechanism of a secure system. As such, they must be easy to access. Although the entity's Privileges API provides sufficient mechanisms to extract the required information, there must be additional functionality provided in order to extract individual fields from within a record. Figure 29 defines the API for accessing and manipulating the various privileges of an entity. In Figure 29 <priv> refers to one of *Groups*, *Associations*, *Level*, *Categories*, or *Caveats*.

The granularity provided by the above functions is rather coarse. However, *Idyllic* provides a rich environment in which to extend the syntax by means of syntactic sugar. These extensions will be highlighted in Chapter 4 when the GPE is illustrated by implementing a few well-understood security problems.

3.5.7 Access Controls API

The access controls subsystem is actually divided into two distinct pieces: the data storing the information and the API providing the access routines. Since the entity stores all relevant information so as to be applicable to *any* security policy, the authors of a particular security policy must create appropriate helper functions that retrieve and

manipulate the entity information. In the case of Bell-LaPadula, the helper functions would focus on retrieving classification levels for the mandatory controls. In conjunction with these, it is highly probable that the authors would provide functions to perform appropriate evaluations for set inclusion and dominance to further enhance readability. The GPE, however, provides sufficient syntax enhancement to lessen the arduousness of programming in the GPE's base state. It is recommended that anyone utilizing the GPE enhance the syntax as outlined in Chapter 4.

<i>gpeAC</i> (uid put <right> allow uid2)	<i>Modifies uid's access controls to allow uid2 the specified <right>. Returns #T on success, #F otherwise.</i>
<i>gpeAC</i> (uid put <right> deny uid2)	<i>Modifies uid's access controls to deny uid2 the specified <right>. Returns #T on success, #F otherwise.</i>
<i>gpeAC</i> (uid remove <right> allow uid2)	<i>Removes uid2 privilege to access uid via <right>. Returns #T on success, #F otherwise.</i>
<i>gpeAC</i> (uid remove <right> deny uid2)	<i>Removes uid2 explicit denial of access to uid via <right>. Returns #T on success, #F otherwise.</i>
<i>gpeAC</i> (uid get <right> allow)	<i>Returns the allow list for the given <right> for the given uid.</i>
<i>gpeAC</i> (uid get <right> deny)	<i>Returns the deny list for the given <right> for the given uid</i>

Figure 30. Access Controls API

Access controls have a number of elements that must be easy to access. Although the entity's Access Controls API provides sufficient mechanisms to extract the required information, there must be additional functionality provided in order to extract individual fields from within a record. Figure 30 defines the API for accessing and manipulating the various aspects of the Access Controls data elements of the entity data structure. In the following table <right> refers to one of *read*, *write*, *execute*, *copy*, or *delete*.

3.5.8 Security Policy API

The GPE provides for a generic API to access the security policy. This interface consists of a single call *gpe* () defined in Figure 31. Valid actions are summarized in Figure 32.

gpe (entity1 action entity2) *Returns #T if entity1 can perform the indicated action against entity2, #F otherwise.*

Figure 31. GPE Security Policy API

Action

read	<i>On access, read the information protected by the entity. On examine, view the list of entities allowed to read the information.</i>
write	<i>On access, write into the information protected by the entity. On examine, view the list of entities allowed to write information to the entity.</i>
execute	<i>On access, execute the information protected by the entity. On examine, view the list of entities allowed to execute the entity's information.</i>
copy	<i>On access, copy the information protected by the entity. On examine, view the list of entities allowed to copy the entity's information.</i>
delete	<i>On access, delete information protected by the entity. On examine, view the list of entities allowed to delete the entity's information.</i>
grant	<i>To which entities can this entity be "given", i.e., granted ownership.</i>

Figure 32. GPE API Commands

There must be two delineated access types: to the entity and to the data protected by the entity. In the former, access controls manipulation and examination is performed without accessing the data protected; examples from typical security includes changing the read and write permissions to a file or device. The latter represents actual access requests to the information protected by the entity. Although the Generic Policy Engine does not store the actual information, it acts as arbitrator on behalf of the calling application. Of the five actions allowed by the Generic Policy Engine Security Policy API, only the *access* action refers to the information itself and it is the *access* action which is coded as the customized security policy for each application.

This delineation is vital since the Generic Policy Engine does not store the actual information being protected but merely the encapsulator and mediates information requests from the calling application.

The following table enumerates the valid calls to the GPE for invocation of the security policy. As can be seen, the invocation is a simple **entity x action x entity** triple which can map elegantly to any security policy. The simple invocation also allows for easy integration into procedural languages, such as C/C++ or Pascal.

<i>gpe</i> (<i>entity1 read entity2</i>)	<i>According to the security policy, can entity1 read entity2?</i>
<i>gpe</i> (<i>entity1 write entity2</i>)	<i>According to the security policy, can entity1 write entity2?</i>
<i>gpe</i> (<i>entity1 execute entity2</i>)	<i>According to the security policy, can entity1 execute entity2?</i>
<i>gpe</i> (<i>entity1 delete entity2</i>)	<i>According to the security policy, can entity1 delete entity2?</i>
<i>gpe</i> (<i>entity1 copy entity2</i>)	<i>According to the security policy, can entity1 copy entity2?</i>

Figure 33. Enumerated GPE Security Policy Function Calls

These calls work for any security policy that can be broken down into one entity requesting access to another through the provided actions (read, write, execute, delete, and copy). These requests for access to information can then be arbitrated by the Generic Policy Engine.

The security policy in the GPE is not integral to the interface but rather interprets incoming requests in a particular manner and passes these requests down to the Generic Policy Engine's Security Policy. The customized portion, that dealing with *access*, is defined to handle the nuances of the particular security policy. Thus a GPE written Bell-LaPadula security policy handles mandatory access controls and discretionary access controls internally, accessing entity control information via the access controls API. Other security policies can also be defined to control access to the information by utilizing the

generic nature of the GPE. The subsequent chapter illustrates the generality of the Generic Policy Engine by implementing a few security policies in common use today.

3.5.9 Security Policy Management

There are two distinct forms of security policy: the administrative security policy, discussed in this subsection, and the explicit security policy, discussed above.

The administrative security policy implicitly defines the behaviour of the explicit security policy by outlining exactly who, what, and how the various fields of an entity can be examined and updated. This behavioural security policy centralizes decisions on whether or not the fields can be manipulated into a single locale to simplify and strengthen required special cases. For example, Bell-LaPadula strictly restricts the motion of information. An entity at a given hierarchical level cannot be downgraded. A mechanism must exist to downgrade information for the system to be useful in the real world. This special case must be handled by defining individuals, known as security officers, with the special ability to circumvent the security policy. This power, obviously, must be controlled. In order to ensure the use is minimized, or at best properly controlled, *gpeManage* () contains special code which may circumvent the overall security policy. Any code that must circumvent the explicit security policy must be placed within *gpeManage* ().

gpeManage (entity1 field entity2) Returns #T if entity1 can update the indicated field of entity2, #F otherwise.

Figure 34. GPE Security Policy Management Private API

The fields which are passed to *gpeManage* () are provided in Figure 35.

UniquelIdentifier	Owner	Groups
EntityType	LastModified	Associations
References	LastAccess	Level
Authentication	PurgeRate	Categories
SecurityPolicy	AccessControls	Caveat

Figure 35. GPE Field List

3.5.10 Invocation Guideline

Security operates under a simple assumption: every action must be approved. Once the action has been approved, the specific action can occur. Sometimes the action requires modifying one of the elements of an entity. In these cases, a subsequent call is made via one of the previously defined APIs which perform the manipulation. All of the APIs, except for the Security Policy API, require only the targeted entity's *uid*. The GPE remembers the last access request, the requesting *uid*, the target *uid*, and the status of the request. For a subsequent call to one of the APIs the requester is therefore known. The action can proceed if and only if the status is *#T* and the API provided *uid* is identical to the one remembered by the security policy for the target. If not, the request is refused.

The following pseudo-code shows how the GPE is used in-line to check on the validity of one entity, in this case a user, attempting to access another entity, a file.

```

if gpe (user access file)
then
    readLine (file)
else
    warn 'access denied'
endif

```

The example illustrates a general use of the GPE to control access to a file by a user for a specific access method. If the security policy determines that the access should be

allowed, the *readLine* is invoked; otherwise, a warning is issued. This general form is all that is required to invoke the security policy implemented in the GPE.

Although the security policy could be called for each invocation of the other API calls used in a program, calls are instead handled implicitly by a call to *gpeManager* () by the API function.

3.5.11 Housekeeping Functions

As with any program that must maintain what is, in effect, a database, a number of housekeeping elements and routines are provided to keep the GPE operating smoothly and efficiently. Of particular note is the *References* list which provides pointers back to every entity which refers to a particular entity. This provides the GPE with the ability to rapidly remove references to a particular entity when it is destroyed and no longer required. Although the GPE could operate correctly without a *References* list, it would be inefficient to examine an entire namespace to determine which entities refer to a particular entity.

3.5.12 Object Reuse

Object reuse comes in two parts: the reuse of the physical data storage used by a given entity protected by Idyllic; and, the actual security parameters stored for a particular destroyed entity. The former case must be handled by the underlying application, the latter is part of the garbage collection regimen of Idyllic. It must be guaranteed that the underlying application, prior to reallocation, clears any information stored within the entity. Similarly, the garbage collection routines for Idyllic must clear any security information from the entity prior to reallocation.

In some cases, such as UNIX, information at the memory level is cleared prior to reallocation. Disk files, however, must be cleared using an additional set of software

either at file destruction time or just prior to reallocation. In the cases of other products, it may be necessary to create custom destruction routines to ensure the information is cleared prior to handoff to an underlying operating system or network. Encryption can be used to perform the object reuse either by the calling application or by Idyllic. It is logically infeasible to define interfaces to every possible kind of program to which the GPE may provide security.

In the Generic Policy Engine, the request to delete an entity automatically forces the system to delete the information associated with that particular entity. Prior to reallocation, Idyllic ensures any information within the entity is purged.

3.5.13 Trusted Path

No trusted path mechanisms are provided by the Generic Policy Engine. A trusted path mechanism, if required for the evaluation of the application utilizing the GPE, must be provided for by the application itself since it is a direct user-system interaction. Additional logic in the security policy can be utilized to ensure the appropriateness of the trusted path command request.

3.6 Target Platforms

Security requires that the security mechanism always be invoked. For any application wherein the flow of information from any point to any other point is well known and defined the GPE will work perfectly. Those applications where there is little clear definition as to where the information is supposed to flow and the paths taken, the GPE will be difficult to incorporate. That said, the latter will also be the least likely applications to be secured and even less likely to be evaluated at anything other than the lowest level of trust within any of the evaluation criteria currently available.

Thus, the GPE is based on the premise that it will be embedded within a properly modular system with clearly defined and controlled flows of control against which policies implemented in the GPE can operate. *Idyllic* and its associated Generic Policy Engines do not rely on any specific operating system or application type. The only requirement is that the underlying operating system or application provide information to the implemented security policy in a format compatible to the GPE API.

3.7 Testing

In order to properly ascertain whether or not the GPE is functionally equivalent to current inline methods of creating security policies, a series of tests and examinations must be performed which adhere to current security criteria. We have already seen the most critical of these requirements: the reference monitor. Meeting the reference monitor requirements is straight-forward and the GPE was designed to meet them fully. But other considerations must also be addressed. First among these is the need to perform at a level which closely matches that of existing technology. If the GPE provides superior security functionality, but at greatly diminished performance, it may not be a practical solution to current security concerns.

Size is also a concern. An application utilizing the GPE must not be much larger than a similarly protected system using an inline approach. If the GPE requires too large an overhead, the usefulness in some environments diminishes. If the GPE cannot function within a good number, if not all, environments equally well, it cannot be truly a *Generic Policy Engine*.

The interface must also be tested. The interface must be seamless and function properly from outside the *Idyllic* environment to be fully useful. The interface should be the only method by which an application can manipulate GPE controlled data.

Similarly, the TCB boundaries must be enforced and protected by the GPE. It must be shown that the boundaries exist and are well defined. The data that is protected by the GPE must be defined clearly and controlled accordingly. The GPE must be able to operate in its own domain, free from interference from rogue system programs.

Chapter 4 provides examples utilizing the GPE and addresses the testing concerns in turn.

Chapter 4

Demonstration & Evaluation

Can a generic security policy engine make general-purpose security practical? In this section we evaluate three security models as implemented using the Generic Policy Engine, place them in context with implementations of existing products offering similar security functionality, and examine the performance of the Generic Policy Engine in comparison to real world applications. The two models chosen reflect common security requirements: a confidentiality model and a message trusted guard model. The confidentiality model will be the Bell-LaPadula security policy model because it is the best known and most frequently implemented of all confidentiality models. The message trusted guard model reflects an amalgam of guard models that protect outgoing messages against improper routing. Both utilize the GPE to implement the security policy.

4.1 Syntactic Sugar

The Generic Policy Engine provides core functionality for implementing security policies. It is not concerned with the vagaries of each and every security policy but rather provides the basic elements required for any security policy. To that end, the functionality provided is spartan. Since Idyllic is a lambda-based language, extending the syntax is a straightforward exercise; syntactic extensions are used to address the readability shortcomings of the Generic Policy Engine's base functionality. This chapter presents a number of security policies which, in order to make the security policies more readable, are provided with their own system calls, their syntactic sugar. The syntactic sugar is a syntax sweetener that allows for the creation of security policy specific extensions.

4.2 The Bell-LaPadula Security Policy Model

Bell and LaPadula [15] defined their security policy in terms of subjects (active entities) and objects¹³ (passive entities). In the original definition of the security model, subjects were processes and objects were files associated with an operating system. For our purposes we shall define the Bell-LaPadula security policy model in terms of their classical use where subjects define active users and objects define passive files and peripherals, such as tape drives and printers.

The Bell-LaPadula Model is described in terms of a simple triple containing the user (or *subject*), the data (*object*), and the action. The two governing rules of the Bell-LaPadula Security Policy Model are presented in Figure 36.

Simple Security Property:	Also known as the no read up (NRU) rule, states that a subject with security label L_S can only read information of an object with security label L_O if and only if L_S dominates (is greater than) L_O .
*-Property:	Also known as the no write down (NWD) rule, states that a subject with security label L_S can only write information to an object with security label L_O if and only if L_O dominates L_S .

Figure 36. Bell-LaPadula Security Properties

Each entity controlled by the Bell-LaPadula security policy model retains state. As a state transition model, states are defined by the set of current accesses, access permissions, assigned security levels, and current hierarchy. Rules define valid state transitions, called axioms, see Figure 37, and these rules must be respected to disallow transitions that lead to an unsecure state. The axioms provided are non-mathematical equivalents to those defined by Bell and LaPadula [15]. The only significant change from Bell-LaPadula is

¹³ Throughout this section we revert to the term "object" in order to remain true to the original work of Bell and LaPadula. As this section is self-contained little confusion should arise. The subjects and objects defined here-in will be termed "active entities" and "passive entities" in subsequent sections to avoid confusion.

that Axiom 5 has been extended to better reflect the least privilege principle and more fully integrate the notion of least privilege into the security policy. For a full description, including underlying mathematics, of the policy model, the reader is referred to [15].

Axiom 1: The Simple Security Policy

The maximum security level of a subject observing an object must dominate the security level of the object.

Axiom 2: The Star Property

The flow of information can only go from a lower security level subject to a higher security level subject.

Axiom 3: The Discretionary Property

The access mode of an accessing subject must be a subset of the set of access permissions that the accessed object grants to that subject.

Axiom 4: The Compatibility Property

All non-directory objects must have the same security level as the directory in which they are contained. The security level of directory objects must dominate the security level of their parent directory. Therefore, the security level can only go up as one travels away from the root of the file hierarchy.

Axiom 5: The Tranquility Property

Only a privileged subject can perform a privileged action such as changing the security level of an object.

Axiom 6: Activation

An object is either active or non-active. A non-active object cannot be accessed by any subject. The state and contents of a newly activated object must be independent from the state and contents of other objects.

Figure 37. Bell-LaPadula Axioms

Implementing Bell-LaPadula in the Generic Policy Engine is straightforward. Subjects become active entities while objects become passive entities. In actuality, subjects would be defined strictly as processes in the calling application though with this implementation of Bell-LaPadula it does not necessarily have to be so.

There are many aspects of the policy model that presuppose specific attributes existing prior to the security policy coming into force. Three of the most crucial are identification and authentication: (I&A), trusted path, and audit mechanisms. These are typically deemed outside the realm of the security policy. The Generic Policy Engine, however, provides a unified approach to security and hence an integrated solution to I&A and audit. These will be utilized in the implementation to provide necessary identification and accountability information necessary for a properly defined security solution. Trusted path is attainable only by the calling application and cannot be completely addressed by the Generic Policy Engine, it is not implemented below. Object reuse requires corresponding code in both the calling application and the GPE. Within the GPE, object reuse is implicitly handled by housekeeping functions such as garbage collection.

4.2.1 *Graphic Representation of Bell-LaPadula*

The Bell-LaPadula security policy model is known as an hierarchical model. This means that the information is segregated into distinct, hierarchical levels each separate and distinguishable from all others and each defined via a dominance relationship with the others. Typically the hierarchy consists of the military classifications *Top Secret*, *Secret*, *Confidential*, and *Unclassified*. They are related in that each in turn dominates the next in a simple mathematical relationship:

$$Top\ Secret > Secret > Confidential > Unclassified.$$

This allows for simple arithmetic rules to be utilized to determine whether an active entity can view an entity by what Bell and LaPadula called *dominance relationships*. For example, an entity at Top Secret would be invisible and unavailable to an active entity at Secret; conversely, an entity at Confidential would be visible and available to the same active entity.

But the Bell-LaPadula security policy model has other aspects, namely categories and caveats that are *not* hierarchical in nature. A given entity is not bound to a single category or caveat as they are to a single level of a hierarchy. Categories and Caveats are viewed as sets with any entity within the Bell-LaPadula security policy model belonging to a predefined set. These categories and caveats are, typically, available to each level and to all entities though not all categories or caveats may be utilized for all entities. Namely, each entity is allowed a specific, predefined subset of the category and caveat sets. Each entity is then restricted to any and all subsets of that predefined subset, including the null set. Hence, the Bell-LaPadula security policy can be graphically illustrated as shown in Figure 38.

Figure 38. Bell-LaPadula Hierarchy Model Graphically Illustrated

What of the case when an entity at a given level, say Secret, and another at Top Secret, do not share the same category and caveat sets? In this case the Bell-LaPadula model becomes a lattice or graph forming what is commonly referred to as a *Hasse* diagram. In such diagrams the vertices of the lattice or graph are connected by lines that show the dominant elements at the top of the diagram and subordinate elements at the bottom. Therefore, as illustrated in Figure 39, the *Top Secret* ([cat 1] [cav 1]) element at the top of the diagram *dominates* the *Top Secret* ([] []) element at the bottom of the diagram.

Figure 39. Hasse Diagram of Two Objects

If we expand the example and include additional categories: *Top Secret* ([cat 2] []) and *Top Secret* ([cat 1 cat 2] [cav 1]) we arrive at Figure 40, a slightly more complex but still manageable arrangement. The diamond shape typifies what one would expect a lattice to resemble since the most dominant element resides at the top and the least dominant at the bottom. Any diagram that does not have unique top and bottom points are not true lattices. There must be an unbroken path from any element in the lattice to the top and bottom elements which is either constantly increasing in dominance or constantly decreasing.

Figure 40. Hasse Diagram of Four Objects

Figure 41. Hasse Diagram of Eight Entities

More complex Hasse diagrams are possible, such as the one depicted in Figure 41, which illustrates the addition of the following elements to the previous diagram:

- *Secret* ([cat 1 cat 2] [cav 1])
- *Confidential* ([cat 2] [])
- *Confidential* ([cat 1] [])
- *Unclassified* ([] [])

As evidenced from Figure 41, as elements are added the complexity of the lattice increases. The complexities increase with every additional element rendering the utilization of diagramming techniques such as Hasse only mildly useful. However, it does assist us in appreciating the complexities of the Bell-LaPadula security policy model and what the Generic Policy Engine must maintain in order to ensure information is properly protected.

4.2.2 Entity Creation and Destruction

Bell-LaPadula models an active set of processes, one where information is moved from one secure or trusted state to another. In order for information to be moved between states it must first be created. If it is to be created, there may come a time when it needs to be destroyed. The Generic Policy Engine provides mechanisms to create and destroy entities and their associated linkages and information. For this implementation of the Bell-LaPadula security policy model we will require two functions as defined below.

createEntity (type entity
classification)

Creates one of the two types of entity required by the Bell-LaPadula security policy (subject or object). The classification level for the new entity is passed as a parameter. On successful creation #T is returned, #F otherwise.

destroyEntity (entity)

Destroys the entity and removes all references within the GPE namespace to it. Returns #T on successful destruction, #F otherwise.

These two functions simply call the existing internal entity creation and destruction routines. All housekeeping functions are handled implicitly by the Generic Policy Engine leaving the security policy uncluttered by such matters.

4.2.3 *The Least Privilege Principle*

The least privilege requires that the person performing a duty must be granted only those system privileges necessary to properly perform a particular duty. This is contrary to many systems, such as UNIX, that provide all or nothing privileges under the guise of a super user or administrator. The typical solution is to create privileges that ensure a particular task requires a particular privilege. Therefore, the tasks associated with an operator – mounting tapes, starting and stopping the printer queues, routine non-security related maintenance – would require *operator privilege*; the creation or modification of user accounts, upgrade or removal of applications, etc. would require *system privilege*. The privileges are typically associated with *roles* such as *operator* or *system administrator*. Applying this solution to UNIX would result in at least three distinct user types: unprivileged user, operator, and system administrator. Security officer is typically added and is responsible for the system logs, maintenance of the security policy, security policy enforcement, and information flow, typically in the form of document downgrades.

Each entity must be assigned a privilege set at creation time. The privilege set is used to determine which functionality the entity can utilize during its operation as an active entity. Least privilege further stipulates that the entity activate only those privileges required to perform its duties.

The Bell-LaPadula model does not require the inclusion of privileges, relying instead on the division of "subjects" and "objects" according to the defined hierarchy. By defining system applications as "System Low", i.e., below all other levels of the hierarchy, sensitive files can be protected from inadvertent modification. By defining unique

categories specific to the various system administrative duties, specific tasks can be delegated to specific individuals. In this way we can subdivide all users into one of two types: privileged users and unprivileged users. We can further subdivide each of these groupings as required by allocating specific categories to differentiate specific privileged sets. For example, categories could be defined to represent the system administrator, operator, and security officer. We shall forgo the use of privileged users for the purposes of this example as they are not necessary to illustrate the Bell-LaPadula security policy.

4.2.4 *Mandatory Access Controls*

Mandatory access controls (MAC) consist of axioms 1, 2, and 4 of the BLP model. MAC provides system administrators the ability to control access to entities by active entities in a mandatory (system defined) way. These controls are regarded by some as system administrator discretionary controls over the entire system. MAC controls typically compartmentalize information. The most common use is to impose hierarchies upon the information such as unclassified, confidential, secret, and top secret. The rules governing MAC define groupings of information to which a active entity *must* belong *prior* to access. The GPE provides for *Levels* which are used to compartmentalize entities to better control information flow through the system. The compartmented entities are usually further subdivided by means of categories and caveats. The definition of valid sensitivity levels, categories, and caveats must be defined prior to the first utilization of the GPE. This is accomplished by defining each valid level, the explicit hierarchy they represent, and the valid category and caveat sets.

Using some syntactic sugar that creates and populates appropriate entities for us, we first create each hierarchical level.

```

(createLevel 'SystemHigh)
(createLevel 'TopSecret)
(createLevel 'Secret)
(createLevel 'Confidential)
(createLevel 'Unclassified)
(createLevel 'SystemLow)           ;* where the system files live

```

Once all the levels have been created we create the explicit hierarchy. Idyllic allows us to create the hierarchy simply as a list. This list explicitly defines the hierarchy and allows for simple arithmetic operations to be utilized to determine which security level dominates another relative to position in the list; the higher the position, the higher the level.

```

(define BLPHierarchy (LIST SystemHigh
                             TopSecret
                             Secret
                             Confidential
                             Unclassified
                             SystemLow))

```

Categories and caveats are unordered sets and are similarly created. Instead of creating a hierarchy, they represent valid sets.

```

(createCategories      'NATO)
(createCategories      'NUCLEAR)

(define BLPCategories (LIST NUCLEAR NATO))

(createCaveats         'CanadianEyesOnly)
(createCaveats         'PrivyCouncilOnly)

(define BLPcaveats     (LIST CanadianEyesOnly PrivyCouncilOnly))

```

Every entity begins with the following set as the default when created:

```

((SystemLow SystemLow SystemLow)  ([])  ([])  // Categories
                                ([])  ([])) // Caveats
  Current      Min      Max      Current Fullset
           Levels

```

This triple defines the entity at *SystemLow*, and has empty sets for both the category and caveat lists. One of the first actions performed is to upgrade the entity to its appropriate current, minimum, and maximum levels, which in our example would be anything from *Unclassified* through *TopSecret*; the GPE would also insert the appropriate caveats and categories for the current active set and the full set.

Therefore, an entity with a security level of:

((Confidential Confidential Secret) ([] []) ([] []))

is said to dominate one with the following level:

((Unclassified Unclassified Confidential) ([] []) ([] []))

since the current hierarchical level of the former is *Confidential* and that of the latter *Unclassified*, regardless of how high or low any given entity may reside.

MAC System Calls

Two system calls are typically required to utilize MAC: *setMAC* () and *getMAC* (). The MAC settings are usually defined as ranges:

SecurityLevel = [min, max]
 Categories = [ct₁, ct₂, ..., ct_N]
 Caveats = [cv₁, cv₂, ..., cv_N]

setMAC (entity classification)	Sets the security level, categories, and caveats for entity to the classification information provided.
getMAC (entity)	Returns the classification information for entity as a triple. If entity doesn't exist, the request is refused.

4.2.5 Discretionary Access Controls

Discretionary Access Controls (DAC) correspond to Axiom 3 of the Bell-LaPadula model. DAC allows system users to restrict access of the entities that they own to specific active entities, in a discretionary way. The TCSEC requires at C2 and above that “access controls ... be capable of including or excluding access to the granularity of a single user” [54, pg. 15]. Some operating systems, most notably UNIX, do not support such permissions via self/group/public permission bits. True access controls follow those outlined in Multics [44] and implemented by the Generic Policy Engine.

Permissions come in two primary flavours: role based and access control based. Both are very similar with role based access control defining what specific roles can or cannot access particular information and having all users within the system defined by specific roles. Strict access controls define specific users or groups of users as having access to particular information. Both are equally valid. For our purposes, we will use access controls since they more closely correspond to a majority of the Bell-LaPadula interpretations currently in use throughout computer security. Permissions are typically defined by read, write, and execute controls.

For our purposes we will define access controls as lists of the following form:

```
(deny (. . .))  
(allow (. . .))
```

for each of read, write, and execute controls. The full access control list (ACL) for any given entity controlled via the Bell-LaPadula security policy would have the following general format:


```

(read      (deny (. . .))
            (allow (. . .)))

(write     (deny (. . .))
            (allow (. . .)))

(execute   (deny (. . .))
            (allow (. . .)))

```

Therefore, an ACL for a given entity of the form:

```

(read      (deny ( ))
            (allow (engineering eugen)))

(write     (deny (engineering))
            (allow (eugen)))

(execute   (deny ( ))
            (allow ( )))

```

would permit engineering and *eugen* to read the contents of the entity while denying all others¹⁴. Similarly, although engineering is explicitly denied write access, eugen is explicitly allowed access. If *eugen* was also a member of engineering the specific granting of a privilege *always* supercedes the general denial, and vice versa.

This general ACL structure allows us to create any variation of permissions desired for any application, be it an operating system or otherwise. This differs from the mechanisms typically employed by trusted UNIX implementations, such as Trusted Xenix [26] and HP-UX [32] in that they employ a fixed size ACL (1 kilobyte per file) and utilize a triple (user, group, access permissions). Tanenbaum and Mullender provided a study in 1984 that showed most UNIX files to be approximately 1k in size. Although this has most

¹⁴ The common practice in computer security is to deny access unless explicitly granted. The reasoning is that it is much easier to forget to deny someone access than for someone who is supposed to have access to not request it, and quite vehemently at times. The reason for the inclusion of "deny" in the ACL format is to allow for explicit denial of access to particular groups who may *otherwise* have been granted access by inclusion in a supergroup. For example, if one wished to deny John read access but John was in the engineering group, there would be no simple mechanism to accomplish this. However, explicitly denying John overrides the explicit allowing of engineering staff to read a given entity. Specific granting or denial of access supercedes more general granting or denial.

assuredly increased, utilizing a fixed ACL size of 1k will still amount to utilizing a good proportion of available disk space on the ACL when most files will not have complex ACLs.

DAC System Calls

In order to manipulate the access control lists a number of system calls are required. The following table summarizes the three functions which provide the Bell-LaPadula security policy with the functionality required to manage the ACLs. Although not explicitly mentioned, the Bell-LaPadula security policy model requires mechanisms by which the ACLs can be updated, removed, and fetched.

<i>removeACL</i> (<i>entity access privilege entity2</i>)	<i>Access can be one of deny or allow. Entity2 is removed from the provided privilege in the ACL.</i>
<i>modifyACL</i> (<i>entity ACL</i>)	<i>Modify an existing ACL by replacing the ACL with the one provided.</i>
<i>getACL</i> (<i>entity</i>)	<i>Retrieve the access control lists for entity. If entity doesn't exist, return #F otherwise the ACL.</i>

4.2.6 Object Reuse

As entities are dynamically created and destroyed, the issue of object reuse must be addressed. Object reuse provides for the return of information to the system's free pool allowing the space occupied to be reused by other applications. Any system that uses Idyllic to implement and maintain its security policy divides the issue of object reuse into two distinct types: Idyllic object reuse and system object reuse. Figure 42 illustrates what happens when there is a request for the deletion or creation of a particular data file within the system. When a request is received by the system for the destruction of a particular data file, the system must ensure that the requesting entity has appropriate access. If all controls are correct, then the entity in question can be destroyed. This destruction is twofold: First, the instance representing the entity within the security policy under Idyllic

must remove the entity and all references to it. Once that process has successfully completed, the system itself can remove the physical storage for the entity and, if need be, sanitize it¹⁵.

Figure 42. Dual Requirement for Object Reuse at the System and GPE Level.

All security criteria require some form of object reuse that ensures that any reallocated entity's contents are independent from its previous instance. Some operating systems, such as UNIX, clear all memory prior to reallocation; others, such as DOS, do not. Therefore, depending on the operating system or application using Idyllic, it may be necessary to create an object reuse function to properly sanitize entities returned to the free memory pool.

4.2.7 Identification and Authentication

In order for the Bell-LaPadula model to operate a mechanism of identification and authentication must be in place. The GPE provides identification and authentication functionality fully capable of delivering sufficient uniqueness to describe all operational

¹⁵ In some instances the sanitization can occur *prior* to reallocation; in other instances it may be preferable to perform the sanitization at deallocation time.

active entities within the model. For the purposes of this example a simple login ID and password suffices to properly and uniquely identify each active entity to the security policy. In order to facilitate the usage, the following routines are provided which provide I&A mechanisms for the Bell-LaPadula security policy.

Login (*user password
level category caveat*)

Attempts to log in the user with the provided password, level, category, and caveat. On success, #T is returned, otherwise #F.

Logout (*user*)

Logs the user out.

4.2.8 Trusted Path

Many high security systems provide a trusted path mechanism whereby a secured line of communication is set up between the user and the system. As the trusted path attention mechanism is typically hardware oriented, it is outside the Generic Policy Engine. However, the Generic Policy Engine does provide a system call that can be utilized to invoke specific system commands securely by the user.

4.2.9 Audit Mechanisms

The Generic Policy Engine implicitly handles the most common forms of audit found in many trusted systems. Auditable events such as entity creation, entity destruction, modification of an entity's security parameters, etc. are all logged by the Generic Policy Engine. Bell-LaPadula does not explicitly require audit since in a perfectly functioning Bell-LaPadula Security Policy Model, no breaches would be possible. It is only when the implementation moves from the realm of mathematics to one of computer logic that errors can occur that cannot be easily verified. These errors could result in breaches of security with a resulting improper disclosure or manipulation of protected information. Audit facilities are therefore utilized in an attempt to catch breaches as they occur for various reasons including possible prosecution. However, for our purposes and since the Bell-LaPadula security policy does not explicitly require audit, it is sufficient to rely on

what the Generic Policy Engine provides as default – namely, all major entity modifications are logged by the GPE for future perusal by appropriate security personnel.

4.2.10 The Security Policy

Now that the entire infrastructure is in place, we need to define the actual Bell-LaPadula Security Policy. Figure 43 defines the implementation of the Bell-LaPadula Security Policy. Utilizing the syntactic sugar defined, we have avoided using raw GPE calls. This allows for a shorter and more readable Bell-LaPadula Security Policy implementation without diminishing the verifiability provided by the GPE security framework. When writing code with the GPE framework, it is important to remember the primary goals of the GPE, and, in fact, all security policy implementations: small, readable code that is verifiable and efficient. The following definition epitomizes this goal.

```
(define (Bell-LaPadula subject access object)
  (let* (
    (Subject-Level (getLevel      subject))      ;;* MAC info
    (Subject-Cats  (getCategories subject))
    (Subject-Cavs  (getCaveats   subject))
    (Object-Level  (getLevel      object))
    (Object-Cats   (getCategories object))
    (Object-Cavs   (getCaveats   object))

    (Allowed       (getRight object access 'allow)) ;;* DAC info
    (Denied        (getRight object access 'deny)))

    ; in
    (and (or (= Subject-Level Object-Level)      ;;* MAC

            ;;* no read up

            (and (> Subject-Level Object-Level)
                  (eq? access 'Read)
                  (subset? Object-Cavs Subject-Cavs))

            ;;* no write down

            (and (< Subject-Level Object-Level)
                  (eq? access 'Write)
                  (subset? Object-Cavs Subject-Cavs)))

          (and (member subject Allowed)          ;;* DAC
                (not (member subject Denied))))

    ) ) )
```

Figure 43. Bell-LaPadula Security Policy Model in the GPE

The simplicity of the above is a direct outgrowth of:

- the ease with which one can manipulate symbols in a symbolic language such as LISP and LISP-like languages; and,
- the underlying functionality provided by the GPE which provides additional security relevant facilities. These additional facilities provide for the needed security functionality typically lacking in operating systems.

Standard access control lists are not typically defined within the two properties associated with the Bell-LaPadula model even though they are always included within operational implementations of the security policy. For completeness, the above policy definition includes access control lists as defined on many systems.

Now, in the provided prototype the Bell-LaPadula Security Policy uses readable names to differentiate various entities. If the GPE were connected to a working operating system, however, unique identifiers understood by the operating system would be used instead. The GPE requires only that the identifiers be unique, regardless of syntax. Most operating systems use numeric representations to ensure uniqueness. This is sufficient for the GPE.

With the Generic Policy Engine, the entire Bell-LaPadula security policy, typically thousands of lines of system code, has been implemented in less than 10 lines of code. This security policy would be embedded as the default security policy for all entities that were to be governed by the Bell-LaPadula security policy model.

Support Routines

Even though there exists within the GPE and the earlier syntactic sugar routines for accessing the various security aspects of an entity, the security policy proper requires

specific routines which return numeric values for the security information to be able to properly execute the security policy.

The Bell-LaPadula security policy model definition utilizes five functions, as defined below, to assist in manipulating the security information properly and efficiently for the Bell-LaPadula security policy model.

getLevel (entity)	Returns the current security level for the entity as a numeric quantity.
getRight (entity privilege)	Returns the requested privilege as a list for the entity.
getCategories (entity)	Returns the current category set for the entity.
getCaveat (entity)	Returns the current caveat set for the entity.
subset? (set1 set2)	Returns #T if the set1 is a true subset of set2, #F otherwise.

4.2.11 Using the Bell-LaPadula Security Policy

The Bell-LaPadula Security Policy Model controls the disclosure of information. To perform this task in a dynamic environment requires that the access controls be constantly updated with current and relevant information pertaining to valid subject-object (active entity-passive entity) interactions. The following figures illustrate how the access controls can be updated and how the Bell-LaPadula security policy is invoked to mediate subject-object interaction.

gpeAC (ebacic put write allow GPE.doc)	Request that ebacic be allowed to write to GPE.doc
gpeAC (jagoda put read allow GPE.doc)	Request that jagoda be allowed to read GPE.doc
gpeAC (ariana put write deny GPE.doc)	Request that ariana be denied write access to GPE.doc
gpeAC (goran put delete deny GPE.doc)	Request that goran be denied delete privilege to GPE.doc

Figure 44. Updating the Access Controls via the GPE

<code>gpeAC (GPE.doc get read allow)</code>	Returns all entities that are allowed to read GPE.doc
<code>gpe (DOR get execute deny)</code>	Returns all entities that are allowed to execute DOR

Figure 45. Examining Access Controls via the GPE

<code>gpe (ed read GPE.doc)</code>	returns whether ed can read the file GPE.doc
<code>gpe (john write bp.doc)</code>	returns whether john can write to the file bp.doc
<code>gpe (dan delete john)</code>	returns whether dan has the right to delete john

Figure 46. Access Control Mediation by the GPE

4.3 Message Trusted Guard

Message Trusted Guards (MTGs) are defined in terms of users and electronic mail addresses. Controls are placed upon the destinations to which specific users can transmit electronic mail messages. No restrictions are placed on incoming e-mail, even though a Message Trusted Guard is capable of controlling both incoming and outgoing connections.

Modern implementations of Message Trusted Guards are embedded within Firewalls, however, earlier this decade stand-alone MTGs were quite common and typically were thousands of lines of C code, much of which handled housekeeping functions for the security controls. The GPE implementation, described below, is less than 50 lines and offers the same functionality.

Implementing an MTG security policy in the GPE is straightforward with the active entity being the actual e-mail message/sender pair and the passive entity being the recipient/e-mail address pair. The action is always "send". Therefore the calling sequence to the GPE coded security policy is: *gpe (sender messageID recipient)*.

4.3.1 Entity Creation and Destruction

There are two types of entities that are maintained by the Message Trusted Guard: senders and recipients. The most common implementation places restrictions on who can transmit to a given recipient. If a given sender is not explicitly permitted, transmission is denied. In order for the e-mail messages to be transmitted from the current site to a desired destination both the sender and recipient must exist. To exist the entities must be created. There may come a time when an entity needs to be destroyed. We'll utilize the Generic Policy Engine's mechanisms to create and destroy entities and their associated linkages and information. For the Message Trusted Guard we will define two functions.

createEntity (type uid)

Creates one of the two types of entity (site or user) with the provided unique identifier. On successful creation #T is returned, #F otherwise.

destroyEntity (uid)

Destroys the entity and removes all references. Returns #T on successful destruction, #F otherwise.

These two functions simply call the existing internal entity creation and destruction routines. All housekeeping functions are handled implicitly by the Generic Policy Engine leaving the security policy uncluttered by such matters.

All users of a network capable of transmitting e-mail would be registered with the Message Trusted Guard when their accounts and e-mail are initially set up. Recipient restrictions are created by the security officer on an address by address basis. If a given address is not in the Message Trusted Guard's list of restricted addresses, then all traffic can flow to that address. This eliminates the need to have a constantly updated table of Internet sites in the Message Trusted Guard.

4.3.2 Transmission Rights

Whether or not a user can transmit to an address follows a simple rule:

```
if exists (recipientAddress)
then
    return (user not in recipientAddress.deny) and
           (user in recipientAddress.allow)
else
    return True      ;* address is not explicitly indicated - allow
endif
```

By default, the e-mail must go through. Therefore, it is up to the security officer to modify the transmission rights for particular sites. Although this is opposite to the typical "deny unless explicitly allowed" rule of computer security, it makes more sense for e-mail. For particularly sensitive destinations, the security officer should preconfigure the MTG to ensure some semblance of protection is provided.

In order to define those sites which are to be restricted and for which users, the following functions are defined:

<i>removeSite</i> (site)	<i>Remove site from the list of sites to which access is restricted.</i>
<i>addSite</i> (site)	<i>Add site to the list of sites to which access is restricted. Initially the allow and deny lists are empty.</i>
<i>denySite</i> (user site)	<i>Disallow user from transmitting to site.</i>
<i>allowSite</i> (user site)	<i>Allow user to transmit to site.</i>
<i>removeUser</i> (user site)	<i>Remove user from the allow or deny lists (the user can only reside on a single list).</i>

4.3.3 Object Reuse

The GPE handles all object reuse. As entities are destroyed, their space is returned to the system. Upon creation of an entity, the address space for the entity is cleansed and put into the entity space.

4.3.4 *Identification and Authentication*

A Message Trusted Guard requires to identification and authentication. All users must be registered with the Message Trusted Guard prior to being allowed to transmit. If they are not in the Message Trusted Guard's list of users, all e-mail by the user is rejected and the action audited for future actioning by the security officer.

4.3.5 *Trusted Path*

No trusted path is required by the Message Trusted Guard.

4.3.6 *Audit Mechanisms*

The Generic Policy Engine implicitly handles audit for the Message Trusted Guard. All mail messages are audited as the flow through the system by explicitly calling the GPE audit routines. Information stored includes sender, recipient, message ID, and status of the transmission.

4.3.7 *The Security Policy*

Historically, the most difficult aspect of the Message Trusted Guard is all the supporting security software. Utilizing the GPE reduces the surrounding code to a mere fraction of C implementations. Figure 47 defines the implementation of the Message Trusted Guard Security Policy. Utilizing the syntactic sugar defined, we have avoided using the raw GPE calls as much as possible.

```

; * define "send" as "write" to make the security policy
; * that much more readable.

(define send 'write)

; *
; * The security policy proper
; *

(define (Message-Trusted-Guard sender messageID recipient)
  (cond
    ((not (entityExists? sender))
     (define status #F))                                ; * deny send

    ((not (entityExists? recipient))
     (define status #T))                                ; * allow send

    (else
     (let* (
       (Allowed (gpeAC recipient 'get send 'allow))
       (Denied (gpeAC recipient 'get send 'deny)))
      in
      ; * allow only if not explicitly denied,
      ; * but explicitly allowed!

      (define status (and (not (member sender Denied))
                          (member sender Allowed))))

     (gpeLog sender put (list sender messageID recipient status))

     ; * return the status of the send

     status
    )
  )

```

Figure 47. Message Trusted Guard Security Policy Model in the GPE

4.3.8 Using the Message Trusted Guard

The Message Trusted Guard Policy Model controls the transmission of electronic mail between a host site and select destinations. The following illustrates how a calling application can quickly deduce whether or not a message can be transmitted to the destination:

gpe (user@secret.org msgID:983734 dictator@banana.republic)

The function *gpe* () invokes the security policy *Message-Trusted-Guard* () and either a true is received granting transmission rights to the electronic mail message or a false is received, denying transmission rights.

4.4 Flexibility Assessment

The GPE offers a unique and flexible approach to creating security policies. The policies are defined solely as security policies performing a singular task: that of mediating information flow. The security policies implemented utilizing the GPE are short and easy to read guaranteeing fewer errors and omissions. Two example security policies have been provided illustrating the elegance of GPE implemented security policies. These security policies illustrate two popular security policies currently in use.

To illustrate the flexibility of the GPE, Figure 48 provides the code for the Biba Integrity Security Policy utilizing the existing code of the Bell-LaPadula Security Policy. As Biba is the complement of the Bell-LaPadula security policy it should be a simple matter to modify Bell-LaPadula into Biba. As can be seen, the only changes required (as noted) are trivial. Furthermore, the support routines for Bell-LaPadula remain unchanged.

```

(define (Biba subject access object)
  (let* (
    (Subject-Level (getLevel      subject))      ;;* MAC info
    (Subject-Cats  (getCategories subject))
    (Subject-Cavs  (getCaveats   subject))
    (Object-Level  (getLevel      object))
    (Object-Cats   (getCategories object))
    (Object-Cavs   (getCaveats   object))

    (Allowed       (getRight object access 'allow)) ;;* DAC info
    (Denied        (getRight object access 'deny)))

    ; in
    (and (or (= Subject-Level Object-Level)      ;;* MAC

            ;;* no write up

            (and (> Subject-Level Object-Level)
                  (eq? access 'Write)             ;;* CHANGE
                  (subset? Object-Cavs Subject-Cavs))

            ;;* no read down

            (and (< Subject-Level Object-Level)
                  (eq? access 'Read)              ;;* CHANGE
                  (subset? Object-Cavs Subject-Cavs)))

      (and (member subject Allowed)              ;;* DAC
            (not (member subject Denied))))

  ) ) )

```

Figure 48. Biba Security Policy Model in the GPE

4.5 Performance Considerations

The performance of the interpreted GPE embedded within an HTTP Server was 100,000 lines of HTML processed per minute through a simplified Bell-LaPadula Security Policy. There was no noticeable impact observed in the throughput of the server, even though each line had to be processed and approved by the security policy.

Currently the slowest aspect of the GPE is the entity space, which is managed akin to a simple database. Although this could be viewed as a possible trouble spot, there is no reason why a commercial-grade database couldn't be used to manage and maintain the

entities on behalf of the GPE. The performance issue would disappear as would other, less obvious concerns regarding persistence and robustness of the current entity space implementation.

The current implementation of Idyllic is small and interpreted. It offers adequate performance in its interpreted state. However, being small allows it to be easily optimized and the use of a simple language based on Scheme allows for faster, compiled implementations [50]. Once compiled, it is unlikely that Idyllic would not meet the requirements of even the most demanding application.

Chapter 5

Concluding Remarks

This research was motivated by the state-of-the-art security policy creation mechanisms currently in vogue in computer security. The current method of crafting security policies for an imaginary user which is an amalgam of all the vendor's clients does not match the needs of the real world where conformity to predefined security needs does not exist. This definition by low water mark does not address the needs of those requiring highly secure custom security policies. Business rules and security policies are unique per organization and a single security policy cannot begin to address each user's requirements.

Much work has been performed to determine the best way in which to prove security policies correct [17, 21], to model security policies [14, 21], to combine security policies into larger, cohesive wholes [1, 30, 31, 37], and to define security policies themselves [6, 9, 13, 15, 20, 39]. Examination of the verification systems of Verdi and Z provided impetus to produce a language capable of not only defining security policies but verifying them as well. The efforts of Bell and the verification teams for Verdi and Z showed that formalisms based on language semantics were sufficiently powerful to not only model the security of a particular system but to implement it as well.

The security policies developed over the years coupled with recent efforts to combine disparate security policies provided an opportunity to examine the situation from a different perspective – that of a general purpose security application, uniquely suited to the creation and ultimate verification of security policies for any computer application. This Generic Policy Engine formulates high-level semantic definitions of security policies with an emphasis on information flow between secured entities.

5.1 Contributions

The Generic Policy Engine is capable of implementing security policies quickly and efficiently. The current Generic Policy Engine is built on top of a Scheme implementation with specific functionality removed, primarily floating point arithmetic and data types. Extensions to the standard syntax were created using syntax extension mechanisms available within Scheme. These syntax extensions are then reducible via the existing rewrite rules of Scheme to standard Scheme. This provides evidence that the basic verifiability of the Generic Policy Engine is sound.

To avoid confusion with standard Scheme, the language utilized to create the Generic Policy Engine has been called Idyllic. This creation of Idyllic provides further evidence that the final working Generic Policy Engine, complete with Idyllic, would remain small and manageable and provide for the implementation of readable security policies. As a tool for the creation of security policies, it becomes self-evident upon reading the accompanying code for Bell-LaPadula that the security policies are not just easier to read, but simpler and less prone to errors due to complex logic errors. The encapsulation of *just* the security policy logic within the security policy function of the Generic Policy Engine greatly simplifies the code and enhances the quality and clarity.

5.2 Future Work

The Generic Policy Engine has provided an elegant mechanism by which to create security policies. In particular, the removal of non-security policy specific functionality from the security policy allows implementers to focus on the security logic, rather than extraneous functionality. The existing Generic Policy Engine also provides reasonable performance and compatibility with existing Scheme implementations in stand-alone testing. However, several extensions to the Generic Policy Engine are planned that should improve its usefulness.

Performance Considerations

Performance must be tested in real-world situations by implementing the GPE within a legacy system or application. A performance hit on a security conscious systems is inevitable; the goal is to minimize it. The GPE runs quickly, and were it compiled instead of interpreted it would run faster still. The trick is to optimize the GPE without overtly reducing its general usefulness to the problem of creating verifiable security policies consistently. Achieving performance penalties of no more than traditional security solutions, that of a 5% impact, should be possible via standard optimization techniques making the Generic Policy Engine not only an elegant but extremely efficient solution.

Networks and Distributed Systems

As networks become more prevalent, distributed computing will become commonplace. Networks introduce a plethora of concerns not evident with stand-alone systems. Foremost among these is the need to ensure traffic is guaranteed to arrive at its destination without *any* modifications. The GSSAPI presented by [42] provides a baseline upon which to build. As users become more sophisticated and their security requirements increase in complexity, the GPE will provide system administrators with the ability to modify the security underpinnings to meet user demands.

What additions will be required to meet network security requirements of the future? At present we can visualize the need for all of the services offered by the GSSAPI as well as methods of providing link encryption, key management, and file encryption.

It is unclear at the present time what, if any, changes would have to occur to the GPE's current structure to enable it to function in a distributed network environment. One notable addition would be the inclusion of a protocol layer to interact and communicate in an efficient manner with other GPEs on a network. The structure of such a protocol and the changes required of the GPE are left as an open exercise.

Persistence

The implementation of the Generic Policy Engine and Idyllic do not provide industrial strength persistence. In order to utilize the Generic Policy Engine in highly dynamic situations, a persistent store must be created; one in which the central data of the Generic Policy Engine is unlikely to be lost due to application or system failures. This persistent store would be embedded as an underlying caching scheme for Idyllic capable of ensuring that no security data exists which has not been written to permanent storage. Applying existing and emerging technologies from the object-oriented virtual machine world, especially those dealing with distributed object-oriented systems, will provide sufficient technology to achieve the goals of persistence within the Generic Policy Engine.

Formal Proof of Extensibility

Historically, security policies were mathematically modeled either by authors creating a consistent mathematical model and then proving the policy against the model or by using one of a few well defined modeling systems, such as *Gypsy*, *Verdi*, or *Z* [5, 17, 21]. Idyllic is based on Scheme. A denotational semantics of Scheme exists [43] and was used by [29] to prove VLISP correct. As Idyllic is based on the same core language (Scheme 48) and the Generic Policy Engine is no more than syntactic extensions of Scheme, it should be possible to prove these extensions as correct and verifiable. By embedding the prover within Idyllic, automated proofs of security policies would be possible.

Retrofitting the GPE to Legacy Systems

One of the questions that must be asked is whether or not the GPE technology could be retrofitted to legacy systems, such as Unix, Microsoft Windows NT or Novell Netware. Any examination to determine whether or not such a retrofit is possible would require a major amount of research into the various target operating systems. However, it is obvious that one of the major hurdles for any retrofit onto legacy systems would be the GPE requirement for a uniform name space. The defining aspect of any attempt to retrofit

the GPE technology is to what extent would the implementer attempt to control information flow. The definition of the entity space, more than any other, would define the level of security obtainable by utilizing the Generic Policy Engine.

5.3 Summary

I am encouraged by the success of the Generic Policy Engine and the apparent suitability of high-level security policy definitions as a means of defining security policy models. The purpose of the Generic Policy Engine was to create a framework by which it would be easy to create functional and correct security policies for any application. It was imperative that these policies be efficient, small, and easy to understand. The Generic Policy Engine has proven to be such a framework.

Bibliography

- [1] Abrams, Marshall D. and Michael V. Joyce. *On TCB Subsets and Trusted Object Management*. MITRE Technical Report #92W0000248, MITRE, McLean, Virginia. January 1993.
- [2] Abrams, Marshall D. *Perspectives on General TCB Subsets*, Supplementary Reading Material for Tutorial #6, 9th Annual Computer Security Applications Conference, December 7, 1993.
- [3] Abrams, Marshall D. et al. *A Generalized Framework for Access Control: An Informal Description*. MITRE Technical Report #MP-90W00043, MITRE, McLean, Virginia. August 1990.
- [4] Allen, John. *Anatomy of LISP*. McGraw-Hill Book Company, New York, New York. 1978.
- [5] Ambler, Allen L.; Good, Donald I.; Brown, J. C.; Burger, Wilhelm F.; Cohen, Richard M.; Hoch, Charles G.; Wells, R. E. "Gypsy: A language for specification and implementation of verifiable programs." SIGPLAN Notices Volume 12, number 3, March 1977, pp.1-10.
- [6] Amoroso, Ed. "A Model for Denial of Service," *Proceedings of the Computer Security Foundations Workshop III*, Franconia, New Hampshire, June 12 - 14, 1990. pp. 110 - 114.
- [7] Amoroso, Edward G. *Fundamentals of Computer Security Technology*, Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

-
- [8] Anderson, Jim. *Computer Security Technology Planning Study*, ESD-TR-73-51, Volume I, AD-758, ESD/AFSC, Hanscom AFB, Bedford, Massachusetts. October 1972.
 - [9] Basic, Eugen M., "Process Execution Controls as a Mechanism to Ensure Consistency," *Fifth Annual Computer Security Applications Conference*, December, 1989. Tucson, Arizona, USA. pp. 114 - 120.
 - [10] Basic, Eugen M., "The Canadian Trusted Computer Product Evaluation Criteria: Updating the Orange Book with 20/20 Hindsight," *Proceedings of the AFCEA Conference*, April, 1990. Ottawa, Canada.
 - [11] Basic, Eugen M., "Process Execution Controls: Revisited," *Sixth Annual Computer Security Applications Conference*, December, 1990. Tucson, Arizona, USA. pp. 334 - 339.
 - [12] Basic, Eugen M., "The Canadian Trusted Computer Product Evaluation Criteria," *Sixth Annual Computer Security Applications Conference*, December, 1990. Tucson, Arizona, USA. pp. 188 - 196.
 - [13] Basic, Eugen M. and Milan S. Kuchta, "Considerations in the Preparation of a Set of Availability Criteria," *Third Annual Canadian Computer Security Symposium*, May, 1991. Ottawa, Canada. pp. 285 - 292.
 - [14] Bell, D. Elliott. "Concerning 'Modeling' of Computer Security," *Proceedings of the 1988 Symposium on Security and Privacy*, Oakland, Californai. pp. 8 - 13.

-
- [15] Bell, David E. and L.J. LaPadula. *Secure Computer Systems: Mathematical Foundations*, ESD-TR-73-278, Volumes I, II, and III. The MITRE Corporation, March, May, and December 1973.
- [16] Biba, K.J. *Integrity Considerations for Secure Computer Systems*, ESD-TR-76-372, MTR-3153, The MITRE Corporation, Bedford Massachusetts. April 1977.
- [17] Bowen, Jonathan; Gimson, Roger; Topp-Jorgensen, Stig. *Specifying System Implementations in Z*. Oxford: Oxford University Computing Laboratory, Programming Research Group. Technical Monograph PRG-63, 1988.
- [18] Boyer, Robert S.; Moore, J. S. *A Computational Logic*. Academic Press, 1977.
- [19] Brooks, Rodney A., Richard P. Gabriel and Guy L. Steele, Jr. *An Optimizing Compiler for Lexically Scoped LISP*. Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction, pp. 261 - 275.
- [21] Clark, David D. and D.R. Wilson. "A Comparison of Commercial and Military Computer Security Policies," *Proceedings of the 1987 Symposium on Security and Privacy*.
- [21] Craigen, Dan. *Formal Verification Programs: Report 5, Summary Conclusions and Recommendations*. I.P. Sharp Associates Ltd, (Ottawa), Technical Report #5605-81-5, June 1981.
- [22] Communications Security Establishment, *The Canadian Trusted Computer Product Evaluation Criteria*. Version 3.0e, January 1993. The Communications Security Establishment, Government of Canada.

-
- [23] Dybvig, R. Kent. *Three Implementation Models for Scheme*, PhD Dissertation at the University of North Carolina, Chapel Hill. 1987.
 - [24] Feeley, M. and G. Lapalme. "Closure Generation Based on Viewing LAMBDA as EPSILON plus COMPILE," Retrieved from Internet at Département d'informatique et de recherche opérationnelle (IRO), Université de Montréal, Montréal, Québec. August 31, 1990.
 - [25] Gasser, Morrie. *Building a Secure Computer System*, Van Nostrand Reinhold, New York, New York, 1988.
 - [26] Gligor, Virgil, et al. "Design and Implementation of Secure Xenix," *IEEE Transactions on Software Engineering*, Volume 13, Number 2, February 1987. pp. 208 – 221.
 - [27] Goldberg, Adele and David Robson. *Smalltalk-80 The Language and Its Implementation*. Addison Wesley, 1983.
 - [28] Gordon, M.J.C., *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, Berlin, West Germany. 1979.
 - [29] Guttman, J.D., Monk, L.G., Ramsdell, J.D., Farmer, W.M., Swarup, V. "A Guide to VLISP, A Verified Programming Implementation", *The MITRE Corporation Technical Report #M92B091*, September 1992.
 - [30] Hosmer, Hilary H. "Metapolicies I," *ACM SigSAC Special Workshop on Data Management Security and Privacy Standards*, San Antonio, Texas, December 1991.

-
- [31] Hosmer, Hilary H., "Metapolicies II," *Proceedings of the 15th National Computer Security Conference*, October, 1992, Baltimore, Maryland. pp. 369 - 378.
- [32] Hewlett-Packard Company. "Proposal for Discretionary Access Control," *IEEE P1003.6*, March 14, 1988, Austin, Texas.
- [33] *Information Technology Security Evaluation Criteria*. Harmonised Criteria of France - Germany - the Netherlands - the United Kingdom. Version 1, May 2, 1990
- [34] Lord, Tom. *Guile Scheme*, GNU Technical Document, The GNU Project, www.delorie.com/gnu/docs/guile/guile_toc.html.
- [35] Kelsey, Richard A., Jonathan A. Rees. "A Tractable Scheme Implementation," *LISP and Symbolic Computation: An International Journal*, 1994.
- [36] LaPadula, L.J. "Formal Modeling in a Generalized Framework for Access Control," *Proceedings of the Computer Security Foundation Workshop III*, June 12 - 14, 1990. pp. 100 - 109.
- [37] Lee, E.S et al. *Composability of Trusted Systems, Reports 1 - 5*. Computer Systems Research Institute, University of Toronto.
- [38] Lee, Peter and Pleban, U.F., "On the Use of LISP in Implementing Denotational Semantics", *Proceedings of 1986 ACM Conference on LISP and Functional Programming*, Cambridge, Mass., 1986. pp. 233 - 248.

-
- [39] Lee, Theodore M.P. "Using Mandatory Integrity to Enforce "Commercial" Security," *Proceedings of the 1988 Symposium on Security and Privacy*, Oakland, California. pp. 140 - 146.
- [40] Lee, Peter, *Realistic Compiler Generation*. MIT Press, Cambridge, Massachusetts. 1989.
- [41] Lindholm, Tim and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.
- [42] Linn, John. "Generic Security Service Application Program Interface," Usenix Workshop Proceedings, UNIX Security II, August 27 - 28 , 1990. Portland Oregon. pp. 33 - 53.
- [43] Clinger, William (editor), "Revised^{3.99} Report on the Algorithmic Language Scheme", MIT Artificial Intelligence Memo. 1989 Also published as an IEEE standard, P1178.
- [44] Saltzer, J.H. "The Protection and Control of Information Sharing in Multics," *Communications of the ACM*, Volume 17, Number 7, July 1974. pp. 388 - 402.
- [45] Scott, Dana. "Lambda Calculus: Some Models, Some Philosophy", *The Kleene Symposium*, North-Holland, Amsterdam, 1980. pp. 223-265.
- [46] Shalit, Andrew. *The Dylan Reference Manual*, Addison-Wesley Developers Press, 1996.

-
- [47] Shockley, William R., Schell, Roger R., Thompson, M. F., "The Importance of High Assurance Computers for Command, Control, Communications, and Intelligence Systems", 4th Aerospace Computer Security Applications Conference; 1988 Dec 12; Orlando, Florida. pp. 331-342.
- [48] Abelson, Harold, Gerald Jay Sussman and Julie Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, McGraw-Hill Book Company, 1985.
- [49] Slonneger, Kenneth, Kurtz, Barry L., *Formal Syntax and Semantics of Programming Languages: A Laboratory Approach*, Addison-Wesley, 1995.
- [50] Steele, Guy Lewis Jr., *Rabbit: A Compiler for Scheme*, MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.
- [51] Stoy, Joseph. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.
- [52] Strachey, Christopher. "Towards a Formal Semantics", *Formal Language Description Languages*, edited by T.B. Steele, North-Holland, Amsterdam, 1966, pp. 198-220.
- [53] Tanenbaum, Andrew S., *Modern Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1992.
- [54] Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD, December 1985.

-
- [55] National Computer Security Center. *Trusted Database Management System Interpretation*. NCSC-TG-021, Version-1, April 1991.
- [56] Thompson-Foss, Incorporated. *Trusted Tunis Integrity Implementation of Integrity Controls: Reports #1 - #4*. Government of Canada/SSC Contract W2213-9-6367/01SV. September 1989 - January 1990.
- [57] Katzke, Stuart W.; Ruthberg, Zella G., editors. *Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPCIS)*. Washington DC: NIST; January 1989; Special Pub 500-160. SN 003-003-02904-1.

Appendix A

Security Functionality

It is more or less agreed that there exist five distinct security functionalities¹⁶: confidentiality, integrity, availability, accountability, and audit. Many of the subcomponents of the functionalities listed are interdependent, but the groupings allow for a better definition of the types of systems system designers attempt to create. Idyllic and the Generic Policy Engine are capable of explicitly addressing elements of the five functional groups. The prototype security policies implemented in the GPE for confidentiality, integrity, and availability highlight the differences between the groups as well as illustrating the strength and thoroughness of the GPE.

It should be noted that a vocal discussion has been underway for nearly five years attempting to extend upon the five functionality groups listed. This discussion has centred on the issues of networking and distributed computing. The author's views of these discussions are well known in the security arena and are not going to be repeated here. Let it simply be said that the GPE goes a long way in vindicating the author's opinion that the five extant functional groups are sufficient and only require minor revisions to address existing concerns.

A.1 Security Criteria

To be a viable alternative to current security methodologies the GPE must be able to present itself in terms understood by security professionals. The following two sections

¹⁶ For an indepth discussion of the five security functionalities, please refer to the *Canadian Trusted Computer Product Evaluation Criteria* [CTCPEC]. Note that within the CTCPEC the functionalities are defined within *criteria*.

provide information which defines which features of the CTCPEC are implemented by the GPE and to which extent. Further, a discussion as the relative strength of the GPE in terms of the amount of trust that can be placed in such a device is also addressed.

A.2 Functionality Services

may want to break this down via subsections instead of a humungoid table.

The GPE provides for the following functionality as defined by the CTCPEC:

Functionality	Service	Description of Capability
Confidentiality	CD	The ability to completely define a set of discretionary controls on information flow.
	CM	Mandatory controls on information flow can be provided alone or in conjunction with discretionary controls.
	CR	The ability to provide full object reuse on the information it stores on the host operating system's objects. It is, however, the host operating system's job to ensure object reuse for those objects which have been removed from the namespace. Note that the OS would require a single call to remove all references to an object within the GPE. At the point in the OS where this call is made is the point at which object reuse could take place, thus simplifying OS security coding requirements.
Integrity	IB	There remains a requirement to isolate various sensitive portions of the OS from other portions of the OS while also ensuring a domain separation

		<p>between users using the OS and the OS's internals. The GPE, by extension, provides domain separation for its security functionality. The OS can instantiate a copy of the GPE and its associated databases within their own address space thus providing hardware protection from user access. The GPE and its associated databases have no user callable routines.</p>
	ID	<p>The discretionary controls for integrity are the dual of those for confidentiality. Similarly, the GPE is capable of providing full support for all discretionary integrity information flow control.</p>
	IM	<p>As with mandatory confidentiality, the GPE provides the capability of providing mandatory controls alone or in conjunction with discretionary controls. It is further possible to provide a mixture of both confidentiality and integrity controls over information flow at the mandatory and discretionary levels.</p>
	IS	<p>Separation of duties requires that the OS be so designed. The GPE can aid the developers by providing a means of easily checking a user's capabilities against a desired action and by restricting entire groups of information to specific users.</p>
Availability	AC	<p>Rudimentary quotas could be maintained and controlled by a policy written using the GPE.</p>
Accountability	WA	<p>API hooks to audit logs kept by the GPE are provided for use by the host operating system. The</p>

API provides programmers with a completely tested and customizable audit facility. The GPE itself audits its actions via the audit log API.

WI The GPE handles all identification and authentication services. The host operating system need only pass the identification/ authentication pair to the GPE and the GPE will indicate whether or not the pair is valid.

Appendix B

Acronyms & Glossary

Access	The performance of a TCB defined operation on an object.
Access Control List	The data structure associated with some objects to indicate which type of access is granted to one or more subjects for that object.
Access Mediation	TCB determination of authorization and whether access should be granted.
Accountability	The process of ensuring that security relevant events in a product are correctly attributable to a user.
Accreditation	The authorization that is granted for the use of an information technology system to process information in its operational environment.
Administrator	A user to whom an administrative role has been assigned.
Administrative User	<i>See Administrator.</i>
Assurance	The degree of confidence that a product correctly implements the security policy.
Assurance Level	In evaluation criteria, a specific level on a hierarchical scale representing successively increasing confidence that a product implements the security policy.
Authorization	The right by a user or process to obtain a specific type of access to a specific object.
Authorized User	A user to whom authorization has been granted.
Availability	The property that a product's services are accessible when needed and without undue delay.

Bell-LaPadula	The Bell-LaPadula Model. A formal state transition model of computer security policy that describes a set of control rules on the access of objects by users.
BLP	See <i>Bell-LaPadula</i> ..
Common Criteria	The Harmonized Computer Evaluation Criteria (Common Criteria or CC) is a computer security standard developed by Canada, the United States, and the European Union that attempts to harmonize the criteria of the author nations.
Component	An identifiable and self-contained portion of a product.
Compromise	A violation of the product's security policy.
Confidential Export	The term used to denote the encapsulation of sensitive data by encryption so that it may be transmitted or stored on electronic media which would otherwise be unsuitable for sensitive data.
Confidentiality	The property that information is not made available or disclosed to an unauthorized user, process or object.
Confidentiality Bias	The view that a security architecture has been defined to present all security solutions in terms of controlling the disclosure of information.
Covert Channels	Information flows which exist in a product but cannot be controlled through other services.
Criteria	A metric used for the evaluation of the effectiveness of security services provided by an information technology product. A generic term used to describe one of the four modern criteria: The Trusted Computer System Evaluation Criteria (TCSEC or Orange Book), The Information Technology Security Criteria (ITSEC), The Canadian Trusted Computer Product Evaluation Criteria (CTCPEC or Canadian Criteria), and the Common Criteria.
CSE	The Communications Security Establishment.

CTCPEC	The Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) is a computer security standard of Canada.
DAC	Discretionary Access Control. Allows system users to restrict access of objects that they own to specific users in a discretionary way.
Delegation	The passing of authorization from one user or process to another as defined in the product's security policy.
Disclosure	The flow of information from an object to a user or process.
Discretionary	Non-administratively controlled. Under a discretionary policy, authorization and delegation do not require administrative intervention.
DoD	United States Department of Defense.
Domain	The unique context in which a program is operating.
Dominate	A security level L_1 is said to dominate another L_2 if the hierarchical classification of L_1 is greater than or equal to that of L_2 and the nonhierarchical categories of L_1 include all those of L_2 as a subset.
Entity	A generic descriptor used to discuss an object within a product regardless of state.
Evaluated Rating	The rating which a Vendor's product has achieved in a completed evaluation.
Evaluation	The process of achieving assurance given a security policy, a consistent description of the security functions and a targeted assurance level.
Evaluation Authority	The office responsible for approving all technical aspects of the evaluation.
Event	Any action which causes a change in the state of the product.

Export	A flow of information such that the information is no longer under the control of the TCB.
External	Outside of the control of the TCB.
Firewall	A computer system designed to protect corporate networks from unwanted external access, typically from TCP/IP networks such as the Internet.
Formal	Written in a notation based upon well-established mathematical concepts, and accompanied by explanatory informal prose.
Functionality	The totality of the functional services of a product that contributes to security.
Illegal	Unauthorized.
Import	A flow of information such that the information becomes under the control of the TCB.
Individual	See User.
Informal	Written as prose in natural language.
Information Flow	The movement of information between users, processes or objects.
Initialization	Setting an object or a product to a known or predefined state.
Internal	Inside of the control of the TCB.
ITSEC	The Information Technology Security Evaluation Criteria (ITSEC) is a computer security standard of the European Union.
Least Privilege	A design and implementation principle where the granted authorizations for a module, component or process, is equal to the set of authorizations necessary to perform its function.

Level or Level of Service	CTCPEC and Common Criteria specific. A defined and measurable requirement for granularity or strength that addresses a specific set of threats. Each level of service provides a better defence against the threats as the levels increase. Levels of service are hierarchical in terms of protection but not necessarily proper subsets in all cases.
LPP	Least Privilege Principle. Principle requiring that each subject in a system be granted the most restrictive set of privileges needed to perform a given task.
MAC	Mandatory Access Control. Allows the system owner to control the access of objects by users in a mandatory way.
Mandatory	Administratively controlled. Under a mandatory policy, authorization and delegation require administrative intervention.
Mechanism	The algorithm that implements a particular service.
Mediation	The enforcement of a security policy.
Modification	The flow of information from a user or a process to an object.
Object	An encapsulated resource exported by the TCB. A resource which stores or contains information and upon which the TCB enforces mediation.
Object Tag	A tag is a security attribute associated with an object. An object tag can be attached by the TCB to a user, process or object.
Policy	See Security Policy.
Process	An active entity under the control of the TCB.
Process Tag	A tag is a security attribute of a process. A process tag can be attached by the TCB to a user, process or object.
Product	The totality of hardware, firmware, software, and documentation offered by a Vendor for evaluation.

Protected Object	The set of objects which are included within a security policy and considered to be under the control of the TCB.
Prove	A syntax and semantics of a formal notation and a set of proof rules, both defined using a well-established set of mathematical concepts, used to provide a convincing argument.
Rating	The totality of the set of service levels and assurance level of a product. See also Target Rating and Evaluated Rating.
Reference Monitor	An abstract machine concept which mediates accesses to objects by users and processes. A reference monitor embodies three principles: completeness (all accesses are mediated), isolation from interference or tampering, and verifiability.
Resource	A primitive entity exported by or existing in the underlying machine. Anything usable or consumable within a product. See also Object.
Restriction	Limits on access or authorization in the enforcement of the product's security policy.
Security	The quality or state being protected from uncontrolled losses or effects.
Security Critical	Those components of the TCB which directly contribute to the provision of one or more security services.
Security Policy	A set of rules and procedures regulating the use of information including its processing, storage, distribution and presentation.
Security Policy Model	A formal representation of the security policy enforced by the product.
Security Relevant	Those components of the TCB which do not directly contribute to the provision of one or more security services.
Semiformal	Written in a restricted syntax language, and accompanied by explanatory informal prose.

Service (Security Service)	A functional grouping rated for its ability to address a defined set of threats. One or more levels of service is defined for each security service.
Session	A period of time during which a user interacts with the product.
State	Refers to one of the three states an object may be in: user, process, or passive.
Storage	An addressable location to which information can be placed and retrieved.
Subject	The Trusted Computer System Evaluation Criteria (TCSEC) term for a process or user. It is used interchangeably in the TCSEC for both.
System	A set of products integrated to perform an application in its environment.
Tag	A term used to describe any access mediation information associated with users, processes, or objects. The association of a tag with an entity may be explicit or implicit. The tag of an entity is part of its encapsulation by the TCB.
Target Rating	The rating a vendor intends to achieve in an evaluation.
TCB	See <i>Trusted Computing Base</i> .
TCB Boundary	The scope of control to which the TCB maintains enforcement of the product's security policy.
TCSEC	The Trusted Computer Security Evaluation Criteria (TCSEC) is a computer security standard of the United States; or DoD 5200.28-STD.
Trace	A basic or simple correspondence used to provide a convincing argument.
Trusted Computing Base	The hardware, firmware and software involved in enforcing a product's security policy; or the set of security critical components and security relevant components.

Unauthorized	Not authorized. See Authorization.
User	An active entity outside of and not constrained by the product's security policy other than in its interactions with the TCB. The TCB will have explicit and implicit assumptions about users and will use these to create an encapsulated abstraction of the actual entity.
User Tag	A tag is a security attribute associated with a user. A user tag can be attached by the TCB to a user, process or object.
Vendor	The organization offering a product for evaluation and representing the product's interests to the Evaluation Authority.
Verifiable	Trusted systems are verifiable if they can be proven correct by means of formal mathematics either by hand or by using automated provers such as Boyer-Moore, Verdi, or Z.
Violate	Contrary to the product's security policy.

Appendix C

GPE: The Code

The code has been reformatted to fit into the confines of the page.

C.1 The Entity Definition

```
;;*
;;*                               Source code for the Generic Policy Engine
;;*                               *****
;;*
;;* Author:                      Eugen Bacic
;;* Title:                      Entity + Entity API
;;* Version:                    2.0
;;* Date of Creation:          January 25, 1996 (built on earlier
;;*                           functional prototype)
;;*                           Copyright ((c)) 1994, 1995, 1996, 1997
;;*                           Eugen Mate Bacic
;;*
;;* General Note:              The GPE and Idyllic are *not* case
;;*                           sensitive.
;;*
;;* Update History:
;;*
;;*       Prehistory: Created first functional prototypes in
;;*                   Scheme to ensure concepts worked. Not a
;;*                   working version. No longer available
;;*                   except for the API.
;;*
;;*       Jan 25/96: Created the file and added in the stubs
;;*                   for all the functions required for the GPE
;;*                   above and beyond the standard Scheme
;;*                   built-ins.
;;*
;;*       Mar 11/96: Wrote most of the generic code in
;;*                   functional style.
;;*
;;*       Aug 27/96: Rewrote the GPE in an object-based style
;;*                   with embedded policies within each entity
;;*
;;*       Nov 12/97: Standardized entity creation, manipulation
;;*
;;*
;;* The entity macro provides the GPE with the capability
;;* to create entities. Each entity has predefined variables and
;;* methods. A few of the provided methods, such as
;;* SecurityPolicy, can be overridden. Initialization data is
;;* inserted by the GPE at creation time of each entity to ensure
```

```

;;* consistency and accountability.
;;*
;;* The call:
;;*
;;*   (entity 'create <uniqueID>)
;;*
;;* creates an entity as follows:
;;*
;;*   <uniqueID> <=
;;*     (variables (<var> <value>)
;;*       :
;;*       (<var> <value>))
;;*
;;*   (private-methods (message . args)
;;*     ((message) <body>)
;;*     :
;;*     ((message) <body>))
;;*
;;*   (public-methods (message . args)
;;*     ((message) <body>)
;;*     :
;;*     ((message) <body>)))
;;*
;;* This entity is managed by the Idyllic.
;;*
;;* Copyright (c) 1994-97 Eugen Mate Bacic
;;*

;;*
;;* First, let's define the constant data structures for the
;;* entity
;;*
;;* VARIABLES:
;;*   Many of the fields (those tagged with comments) are filled
;;*   in by the define-entity macro.
;;*

(define ENTITY-VARIABLES
  '(UniqueIdentifier EntityType References
    Authentication SecurityPolicy))

  '((UniqueIdentifier '*TO-BE-FILLED-IN*)
    (EntityType 'Entity)
    (References '())
    (Authentication '(*Active* *NoExpiry*
                      *NoPassword*))
    (SecurityPolicy ,defaultSecurityPolicy))

;;*
;;* These are all audit log related. PurgeRate provides a lambda
;;* expression for determining when the audit logs should be purged
;;* and how the purge should occur. This is end-user settable.
;;* Default is the logs are never purged, meaning they grow forever
;;* (i.e., we assume ala Turing, an infinite tapeto store all that

```

```

; * audit information ;-). Each log is of the same format:
; *
; *      (<timestamp> <uid> <status> <info>)
; *
; * where the <timestamp> is the time at which the event was
; * logged,
; *      <uid>      is the entity requesting the action,
; *      <status>   is the status of the request and
; *      <info>     is any additional information the system
; *                  wishes to record concerning the event.
; *
; * The "Private" log is one that can be used by an application to
; * store information. Therefore, if an application wishes to log
; * information particular to its operation, it can utilize an
; * entity for sole purpose of audit. This entity would belong to
; * the calling application and be manipulable solely by the
; * calling application. This alleviates any requirement for the
; * application programmers to create audit level functionality
; * into their code.
; *

(define AUDIT                                '(Owner      LastModified LastAccess
                                                Private PurgeRate))

(define defaultPurgeRate                    (lambda () #f)) ; * never purge
; *
; * The privileges are data elements which are always of the form:
; *
; *      (<currentlyActive> <validElement1> <validElement2> ...)
; *
; * The currently active element is first. Subsequent entries are
; * valid elements which can be set "active". For example, the
; * calling application may set the active Level as Secret while
; * the valid levels for which this entity can access are
; * Confidential, Secret, and Top Secret.
; *

(define PRIVILEGES                          '(Groups      Associations Level
                                                Categories Caveats))

; *
; * For each of the following, we'll create embedded *allow* and
; * *deny* lists. This will be set up when an entity is created and
; * by default allow the *genericEntity* and <creator> of the new
; * entity access controls. The rights granted are for read, write,
; * copy, and delete.
; *

(define ACCESS-CONTROLS                     '(Read Write Execute Delete Copy))

(define defaultPurgeRate                    (lambda () #f)) ; * never purge

(define defaultSecurityPolicy               (lambda (arguments . args) #f))
; * i.e., reject all requests

(define *Active*                            #T)

```

```

(define *NoExpiry*      #F)
(define *NoPassword*    #F)

(define *Deactivated*   #F)
(define *NotActive*     #F)

(define-class Entity
  (constructor-arguments uniqueIdentifier)

  ;* GLOBAL security policy

  (class-variables (SecurityPolicy    defaultSecurityPolicy))

  (class-methods (message . arguments)
    ((setSecurityPolicy)
      (set! SecurityPolicy (argumentList arguments 0)))

    ((setDefaultPolicy)
      (set! SecurityPolicy defaultSecurityPolicy)))

  (slots (EntityType          'Entity)
    (References               '())
    (Authentication           (list *Active* *NoExpiry*
                                     *NoPassword*))
    (localSecurityPolicy      SecurityPolicy)

    (Owner                    '())
    (LastModified              '())
    (LastAccess                '())
    (Private                   '())
    (PurgeRate                 defaultPurgeRate)

    (Groups                    '())
    (Associations              '())
    (Level                     '(*None* *None* *None*))
    (Categories                '(*None* *None* *None*))
    (Caveats                   '(*None* *None* *None*))

    (Read                      '((allow ())
                                (deny  ())))

    (Write                     '((allow ())
                                (deny  ())))

    (Execute                   '((allow ())
                                (deny  ())))

    (Delete                    '((allow ())
                                (deny  ())))

    (Copy                      '((allow ())
                                (deny  ())))
  )

  (methods (message . args)
    ((EntityType)
      EntityType)
  )

```

```

((EntityType:)
  (set! EntityType (car args)))

;*
;* Housekeeping functions, used between entities but not
;* used by external functions -- no external API!
;*

((ReferencesAdd:)
  ;* Add the provided entity as referring to this entity
  'place-holder)
((ReferencesRemove:)
  ;* Remove the provided entity as referring to this entity
  ;* This is usually due to the fact that the other entity
  ;* has removed a reference (i.e., a user no longer has
  ;* access to an entity, or whatever) or the referring
  ;* entity has been removed.
  'place-holder)

;*
;* The following routines handle the authentication
;* mechanisms
;*

((Active)
  ;* Returns whether or not the current entity is active.

  (eq? (car Authentication) *Active*))
((Deactivate)
  ;* Deactivate the entity.

  (replace! Authentication 1 *NotActive*))

((Expires?)
  ;* Returns whether or not the current entity expires.

  (not (eq? (cadr Authentication) *NoExpiry*)))
((Expires:)
  ;* Set an expiry date.

  (replace! Authentication 2 (car args)))

((Password?)
  ;* Returns whether or not the current entity has a
  ;* password.

  (eq? (cadr Authentication) *NoPassword*))
((Password:)
  ;* Sets the password.

  (replace! Authentication 3 (car args)))
((ValidPassword?)
  ;* Checks the validity of a provided password.

  (eq? (caddr Authentication) (car args)))

;*
;* The security policy for all entities can be overridden

```

```

; * via local security policies. These are useful when
; * entities are copied between domains or namespaces.
; *

((resetLocalPolicy)
  (set! localSecurityPolicy #F))

((localPolicy:)
  (set! localSecurityPolicy (car args)))

; *
; * The following functions control the logs
; *

((logEntry:)
  (let (
    (whichLog (car args))
    (entry (list (timestamp) uniqueIdentifier
                  callingUID (cdr args))))
    in
      (case whichLog
        ((Owner) (addToLog Owner entry))
        ((LastModified) (addToLog LastModified entry))
        ((LastAccess) (addToLog LastAccess entry))
        ((Private) (addToLog Private entry))
        (else
         (error "Unknown log '"
                  whichLog
                  "' in logEntry: command."))))))

((getLog)
  (case (car args)
    ((Owner) Owner)
    ((LastModified) LastModified)
    ((LastAccess) LastAccess)
    ((Private) Private)
    (else
     (error "Unknown log '" whichLog
            "' in getLog command."))))

((resetPurgeRate)
  (set! PurgeRate defaultPurgeRate))

; *
; * Privileges, etc. These are handled similarly to Smalltalk
; * collections i.e., at: put: style. Note: Nothing but
; * rudimentary error checking.
; *

((Priv:)
  (let (
    (Index (car args))
    (Command (cadr args)) ; * i.e., put:, remove:
    (Value (caadr args)))
    in
      (case Command
        ((put:)

```

```

      (case Index
        ((Groups)      (insert! Groups      Value))
        ((Associations) (insert! Associations Value))

        ;*
        ;* These are handled differently since they
        ;* are triples. Only one valid level is
        ;* allowed, so we replace it. The others
        ;* we add to.
        ;*

        ((Level)      (replace! Level 1 Value))
        ((Categories) (insert! (car Categories)
                               Value))
        ((Caveats)    (insert! (car Caveats)
                               Value))

        (else
         (error "Unknown priv '" Index
                "' in Priv: put: command.")))

      ((remove:)
       (case Index
         ((Groups)      (remove! Groups      Value))
         ((Associations) (remove! Associations Value))

         ;*
         ;* Level is handled differently since it
         ;* requires a valid setting at all times and
         ;* the null setting is *not* allowed.
         ;* Categories and Caveats are different since
         ;* they are allowed to have one or more
         ;* category or caveat active.
         ;*

         ((Level)
          (error "You cannot remove only replace the
current Level."))

         ((Categories) (remove! (car Categories)
                                Value))
         ((Caveats)    (remove! (car Caveats)
                                Value))

         (else
          (error "Unknown priv '" Index
                 "' in Priv: remove: command.")))

      ((Priv)
       (case (car args)
         ((Groups)      Groups)
         ((Associations) Associations)
         ((Level)      Level) ;* returns the
                               ;* level info,
                               ;* external routines
         ((Categories) Categories) ;* should determine
                               ;* what aspect of each
                               ;* should
         ((Caveats)    Caveats) ;* be returned
         (else
          (error "Unknown priv '" (car args)

```

```

        '' in Priv command.")))))

; *
; * Access controls have a 2-tuple. Depending upon if it is a
; * deny or allow the appropriate list must be updated.
; *

((Allow)      ; * (uid allow uid2 [add | remove]
               ; *      [read | write | execute | delete | copy])
  (let (
    (uid2      (car args))
    (action    (cadr args))
    (right     (caddr args)))
    in
      (case action
        ((add)
          (case right
            ((read)    (insert! (car Read)
                                uid2))
            ((write)   (insert! (car Write)
                                uid2))
            ((execute) (insert! (car Execute)
                                uid2))
            ((delete)  (insert! (car Delete)
                                uid2))
            ((copy)    (insert! (car Copy)
                                uid2))
            (else
             (error "Unknown access right \' "
                    right " \' for add."))))

          ((remove)
          (case right
            ((read)    (remove! (car Read)
                                uid2))
            ((write)   (remove! (car Write)
                                uid2))
            ((execute) (remove! (car Execute)
                                uid2))
            ((delete)  (remove! (car Delete)
                                uid2))
            ((copy)    (remove! (car Copy)
                                uid2))
            (else
             (error "Unknown access right \' "
                    right
                    " \' for remove."))))

          (else
           (error "Unknown action on allow \' "
                  action " \'."))))))

((Deny)      ; * (uid deny uid2 [add | remove]
               ; *      [read | write | execute | delete | copy])
  (let (
    (uid2      (car args))
    (action    (cadr args))
    (right     (caddr args)))
    in
      (case action
        ((add)
          (case right

```



```

((read)      (insert! (cadr Read)
                  uid2))
((write)     (insert! (cadr Write)
                  uid2))
((execute)   (insert! (cadr Execute)
                  uid2))
((delete)    (insert! (cadr Delete)
                  uid2))
((copy)      (insert! (cadr Copy)
                  uid2))
    (else
      (error "Unknown access right '"
              right "' for add.")))

    ((remove)    (case right
                  ((read)      (remove! (cadr Read)
                                          uid2))
                  ((write)     (remove! (cadr Write)
                                          uid2))
                  ((execute)   (remove! (cadr Execute)
                                          uid2))
                  ((delete)    (remove! (cadr Delete)
                                          uid2))
                  ((copy)      (remove! (cadr Copy)
                                          uid2))
                  (else
                    (error "Unknown access right '"
                            right "' for remove.")))

    (else      (error "Unknown action on allow '"
                      action "'.")))

) ) ;* end entity

;*
;* Some helper functions
;*

(define (argumentList arguments n)
  (if (< (length arguments) n)
      (error 'argumentList "Invalid number of arguments")
      (list-ref args n)))
) )

```

C.2 Configuring, Starting, & Stopping the GPE

```

;*****
;
;
;           Global Configuration of the GPE
;*****
;
;
;
; gpeID = gpeCreate ( )
;
; Returns a reference to a new namespace. The application is
; expected to use gpeID whenever it requires access to the
; security functionality. If gpeCreate () could not create a
; new namespace, 0 is returned. gpeID is a long unsigned
; integer. Once opened, all initialization to the global
; status of the GPE must be performed.
;
;

(define (gpeCreate)
  ;* generate unique namespace ID
  ;* return "namespace.timestamp.gpe" as the gpeID

  "namespace"                ;* for now, we don't need to
                              ;* worry about multiple namespaces
)

;
;
; gpeOpen (gpeID)
;
; The application requests a previously opened GPE namespace
; to be opened by calling the function with gpeID. If the
; namespace exists, #T otherwise #F. Once opened, all
; initialization to the global status of the GPE must be
; performed.
;
;

(define (gpeOpen gpeID)
  (load gpeID)                ;* load in the namespace associated
                              ;* with the provided gpeID this
                              ;* should load in all the entities

  (write "GPE namespace ")
  (write gpeID)
  (write " opened.")
  (newline)
  #t
)

;
;
; gpeStart ( )
;
; All initialization and global sets must occur prior to the
; gpeStart () command being issued. This ensures that certain
; attributes are frozen for the running duration of the GPE,
; such as audit level.
;
;

```

[illegible]

C.3 Entity Creation & Control

[illegible]

```

; *
; *
; *      default security policy, the
; *      application should pass
; *      defaultSecurityPolicy as the
; *      parameter. Returns #T on
; *      success, #F otherwise.
; *

```

```

(define (gpeEntity uid command . args)
  (case command
    ((create) (let (
                  (temp (generic-entity create uid)))
                  in
                  (if temp
                     (set! EntityList (cons (list uid temp)
                                              EntityList))
                     temp)))
    ((remove)  ; * track down and remove all references
               ; * remove the entity
               #t)
    ((SP)      ((getEntity uid) setSecurityPolicy (car args)))
    (else      (error "Unknown request to gpeEntity " command)
               #f)
  ) )

```

```

; *
; * If the uid is in the EntityList, it returns the appropriate
; * entity code. Otherwise, it returns a lambda expression of
; * multiple arguments that returns #f (i.e., failure).
; *

```

```

(define (getEntity uid)
  (let (
        (uidExists? (cadr (assoc uid EntityList))))
    ; in
    (if uidExists?
        uidExists?
        ; else
        (lambda (x . moreArgs)
          ; * This is the error return
          ; * value, no such entity!
          ; * it's just a lambda that
          ; * returns #F
          #f)
    ) ) )

```

```

; *
; *****

```

C.4 Identification & Authentication API

```

;*****
;*
;*                               Authentication in the GPE
;*                               *****
;*
;* gpeAuth (uid command [supplementalArguments])
;*
;*   gpeAuth provides authentication mechanisms to the GPE. Only
;*   the following commands are allowed: add, remove, equal?,
;*   expires, expires?, setPasswd, SetPassLen, deactivate, and
;*   activate.
;*
;*   gpeAuth (uid Add authStr)      Adds uid, plus its
;*                                   authentication string
;*                                   (authStr) to the entity
;*                                   referred to by uid. Returns #T
;*                                   on success, #F otherwise.
;*
;*   gpeAuth (uid eq authStr)       Is the uid and authentication
;*                                   string (authStr) correct?
;*                                   #T if correct, #F otherwise.
;*
;*   gpeAuth (uid Expires date)     Sets the expiry date for uid.
;*                                   #T on success, #F on failure.
;*
;*   gpeAuth (uid Expires?)         Returns #F if the uid's
;*                                   password never expires, the
;*                                   date of expiry otherwise.
;*
;*   gpeAuth (uid SetPaswd old new) Sets the password to the new
;*                                   one provided the old password
;*                                   matches the existing one.
;*                                   Returns #T on success, #F on
;*                                   failure.
;*
;*   gpeAuth (uid SetPassLen newlen) Sets the password length to
;*                                   the new one provided so long
;*                                   as it is between the min and
;*                                   max password lengths. Returns
;*                                   #T on success, #F on failure.
;*
;*   gpeAuth (uid deactivate)       Deactivate the uid. #T on
;*                                   success, #F otherwise.
;*
;*   gpeAuth (uid activate)         (Re)activates the uid. Returns
;*                                   #T on success, otherwise #F.
;*
(define (gpeAuth uid command . args)
  (case command
    ((add) (let (
                (temp (generic-entity create uid)))
              in

```

```

                                (if temp
                                  (set! EntityList (cons (list uid
                                                              temp)
                                                            EntityList)))
                                temp))

((eq)                          (equal? ((getEntity uid) getAuthString)
                                         (car args)))

((expires)                     ((getEntity uid) setExpiry (car args)))

((expires?)                    ((getEntity uid) getExpiry))

((setPasswd)                   ((getEntity uid) setPasswd (car args)
                                         (cadr args)))

((SetPassLen)                  ((getEntity uid) setPassLen (car args)))

((deactivate)                  ((getEntity uid) deactivate))

((activate)                    ((getEntity uid) activate))

(else                          (error "Unknown request to gpeAuth "
                                         command)
                                #f)

) )

; *
; *****

```

C.5 Audit API

```

; *****
; *
; *                               Audit in the GPE
; *                               *****
; *
; * Audit requires a set of variables to ensure that the entire
; * system (all entities) are audited at a consistent level. This
; * global setting is changeable *only* prior to gpeStart.
; *

(define AuditLevels
  '(MinimalAuditLevel          ; * The only auditing done is
                               ; * the bare minimum, namely
                               ; * recording entity creates
                               ; * and destroys.

    AuditLevel1                ; * Same as MinimalAudit but
                               ; * with the inclusion of open
                               ; * entity and close entity
                               ; * requests.

```

```
AuditLevel2                /* Same as Level1 but with the
                           /* inclusion of modifications
                           /* (write) to an entity.

    AuditLevel3              /* Same as Level2 but with the
                           /* inclusion of access (read)
                           /* to an entity.

    AuditLevel4              /* Same as Level3 but with the
                           /* inclusion of copies so as
                           /* to track if information is
                           /* being moved from user to
                           /* user via copies rather than
                           /* standard reads.

    MaximumAuditLevel))     /* Same as Level4.
```

```
(define *LogDefaultLevel* 'MinimalAuditLevel)
```

```
;*
;* There are two forms of the gpeLog command:
;*
;*      gpeLog (command [supplementalArguments])
;*
;* which is used to manipulate the global log attributes. These
;* are only accessible during startup and must be used prior to
;* issuing a gpeStart command.
;*
;* The other form is:
;*
;*      gpeLog (uid command argument)
;*
;* which allows for the placing and retrieval of entries from the
;* private log found within each entity. Useful for application
;* specific logging data. The only commands allowed are: get and
;* put.
;*
;*      gpeLog (GetLevel)          Return the currently set level
;*                                of audit.
;*
;*      gpeLog (SetLevel level)   The GPE provides predefined
;*                                levels of granularity
;*                                for audit. <level> must
;*                                correspond to one of the
;*                                predefined audit levels.
;*                                Returns #T on success,
;*                                #F on failure.
;*
;*      gpeLog (setOptions)        Sets the option list to the
;*                                list provided (future).
;*
;*      gpeLog (SetGMT)           Sets the time to GMT, default
;*                                for Time is local time.
;*
;*      gpeLog (SetLocal)         Sets the time to local time,
;*                                this is the default.
```

```

;*      gpeLog (Time)                      Returns the current setting,
;*                                          either Local or GMT.
;*
;*      gpeLog (uid put data)              Appends data to the private
;*                                          for the entity defined by uid.
;*
;*      gpeLog (uid get expr)              Returns all entries for the
;*                                          entity uid which match expr.
;*                                          expr can be a wildcard, which
;*                                          returns the entire log.
;*
(define *GlobalLogCommands* '(GetLevel SetLevel SetOptions
                              SetGMT SetLocal Time))

(define *LogTimeZone* 'Local)

(define *_LOG_* '()) ;* temporary holding log

(define (gpeLog parml . args)
  (if (memq parml *GlobalLogCommands*) ;* gpeLog has a special
      ;* case for global
      ;* commands and those
      ;* which are available
      ;* only during runtime,
      ;* namely those to put
      ;* and get data from the
      ;* logs.
      ;* get is typically
      ;* restricted to
      ;* inactive logs only.
      (define command parml)
      ; else
      (define command (car args))
      )
  (case command
    ((GetLevel) (let (
                    (temp (generic-entity create uid)))
                  in
                    (if temp
                     (set! EntityList (cons (list uid
                                                    temp)
                                              EntityList))
                     temp)))
    ((SetLevel) (set! *LogDefaultLevel* (car args)))
    ((SetOptions) ;* Does nothing yet but return the args,
                  ;* for lack of a better option
                  args)
    ((SetGMT) (set! *LogTimeZone* 'GMT))
    ((SetLocal) (set! *LogTimeZone* 'Local))
    ((Time) *LogTimeZone*)
    ((put) ;* This is temporary. The final version
           ;* will write to an audit log with
           ;* a predefined name, etc. It'll have code
  )

```



```

; * for auto-rollover on a daily,
; * weekly or monthly basis, and minimally
; * impact the system. Full log/
; * file system will be detected and
; * handled. Til then, we just have a
; * consable list...

(set! *_LOG_* (cons (list (timestamp)
                          (currentUID)
                          args)
                    *_LOG_*))

((get) ; * Yet to be done. When finished will
; * accept an expression and return
; * all entries within the log to which the
; * expression matches or closely
; * corresponds. The nature of the search
; * is TBD.

(else (error "Unknown request to gpeLog "
             command)
      #f)

) )

```

C.6 Privileges API

```

; *****
; *
; * Privileges in the GPE
; * *****
; *
; * gpePriv (uid command [supplementalArguments])
; *
; * gpePriv provides access to the privilege mechanisms of the
; * GPE. Only the following commands are allowed: put, remove,
; * and get.
; *
; * gpePriv (uid put priv uid2) Adds uid2 to the privilege
; * specified bypriv for uid.
; * Returns #T on success, #F
; * otherwise.
; *
; * gpePriv (uid remove priv uid2)
; * Removes uid2 from uid's
; * privilege specified by priv.
; * Returns #T on success, #F
; * otherwise.
; *
; * gpePriv (uid get priv) Returns the privilege
; * specified by priv for the
; * given uid.

```



```

; *      gpeAC (uid get right allow)          Returns the allow list
; *                                           for the given <right>
; *                                           for the given uid.
; *
; *      gpeAC (uid get right deny)           Returns the deny list
; *                                           for the given <right>
; *                                           for the given uid.
; *

(define (gpeAC uid command right allow-deny . uid2)
  (if (eq? allow-deny 'deny)
      (case command
        ((put)      ((gpeEntity uid) putAccessRight right deny uid2))

        ((remove)   ((gpeEntity uid) removeAccessRight right deny uid2))

        ((get)      ((getEntity uid) getAccessRight right deny))

        (else       (error "Unknown request to gpeAC "
                           command)
                     #f)
      )
      ; else / Allow
      (case command
        ((put)      ((gpeEntity uid) putAccessRight right allow uid2))

        ((remove)   ((gpeEntity uid) removeAccessRight right allow uid2))

        ((get)      ((getEntity uid) getAccessRight right allow))

        (else       (error "Unknown request to gpeAC "
                           command)
                     #f)
      )
  ) ) )

```

Appendix D

Examples: The Code

The code has been reformatted to fit into the confines of the page.

D.1 Bell-LaPadula Security Policy Model

```
;;*
;;* Syntactic sugar to create levels for Bell-LaPadula.
;;*

(define createLevel Entity)

;;*
;;* define the actual levels to be used (this should only
;;* be added to when the GPE is quiescent, namely at startup
;;* and not ever after!
;;*

(createLevel SystemHigh)
(createLevel TopSecret)
(createLevel Secret)
(createLevel Confidential)
(createLevel Unclassified)
(createLevel SystemLow)      ;* where the system files live

;;*
;;* define the hierarchy for later use, each name (SystemHigh,
;;* etc.) reflects a created entity in the GPE namespace via
;;* the calls above.
;;*

(define BLPHierarchy (LIST SystemHigh
                             TopSecret
                             Secret
                             Confidential
                             Unclassified
                             SystemLow))

;;*
;;* Syntactic sugar for insertion of caveats and categories
;;* into the GPE namespace
;;*
```

```

(define createCategories Entity)

(define createCaveats      Entity)

;;*
;;* define the categories and caveats
;;*

(createCategories      NATO)
(createCategories      NUCLEAR)

(createCaveats         CanadianEyesOnly)
(createCaveats         PrivyCouncilOnly)

(define BLPCategories    (LIST NUCLEAR NATO))
(define BLPcaveats       (LIST CanadianEyesOnly PrivyCouncilOnly))

;;*
;;* BLP support routines
;;*

(define (getLevel entity)
  (assoc (car (gpePriv entity get 'Level)) BLPHierarchy)
)

(define (getCategories entity)
  (car (gpePriv entity get 'Categories))
)

(define (getCaveats entity)
  (car (gpePriv entity get 'Caveats))
)

(define (getRight entity access right)
  (gpeAC entity 'get right access)
)

;;*
;;* The actual security policy. This defines Bell-LaPadula
;;* complete with Mandatory Access Controls and Discretionary
;;* access controls.
;;*

(define (Bell-LaPadula subject access object)
  (let* (
    (Subject-Level (getLevel      subject))      ;;* MAC info
    (Subject-Cats  (getCategories subject))
    (Subject-Cavs  (getCaveats    subject))
    (Object-Level  (getLevel      object))
    (Object-Cats   (getCategories object))
    (Object-Cavs   (getCaveats    object))

    (Allowed       (getRight object access 'allow)) ;;* DAC info
  )

```

```

        (Denied      (getRight object access 'deny)))
;   in
    (and (or (= Subject-Level Object-Level)           ;;* MAC

            ;;* no read up

            (and (> Subject-Level Object-Level)
                (eq? access 'Read)
                (subset? Object-Cavs Subject-Cavs))

            ;;* no write down

            (and (< Subject-Level Object-Level)
                (eq? access 'Write)
                (subset? Object-Cavs Subject-Cavs))))

    (and (member subject Allowed)                      ;;* DAC
        (not (member subject Denied)))

) ) )

```

D.2 Message Trusted Guard

```

;* define "send" as "write" to make the security policy
;* that much more readable.

(define send 'write)

;*
;* Some syntactic sugar to make the MTG calls feel MTG-ish
;*

(define (removeSite site)
  (gpeEntity site 'remove)
)

(define (addSite site)
  (gpeEntity site 'create)
)

(define (denySite user site)
  (gpeAC site put send 'deny user)
)

(define (allowSite user site)
  (gpeAC site put send 'allow user)
)

(define (removeUser user site)
  (gpeAC site remove send 'allow user)
  (gpeAC site remove send 'deny user)
)

```

```

; *
; * The security policy proper
; *

(define (Message-Trusted-Guard sender messageID recipient)
  (cond
    ((not (entityExists? sender))
      (define status #F)) ; ; * deny send

    ((not (entityExists? recipient))
      (define status #T)) ; ; * allow send

    (else
      (let* (
        (Allowed (gpeAC recipient 'get send 'allow))
        (Denied (gpeAC recipient 'get send 'deny)))
        in
          ; * allow only if not explicitly denied,
          ; * but explicitly allowed!

          (define status (and (not (member sender Denied))
                              (member sender Allowed)))))

    (gpeLog sender put (list sender messageID recipient status))

    ; * return the status of the send

    status
  )

```