American Journal of Applied Sciences 5 (9): 1117-1126, 2008
ISSN 1546-9239

Security Policy Development: Towards a Life-Cycle and Logic-Based Verification Model

Luay A. Wahsheh and Jim Alves-Foss
Center for Secure and Dependable Systems, University of Idaho, P. O. Box 441008
Moscow, Idaho 83844-1008, USA

Abstract: Although security plays a major role in the design of software systems, security requirements and policies are usually added to an already existing system, not created in conjunction with the product. As a result, there are often numerous problems with the overall design. In this paper, we discuss the relationship between software engineering, security engineering, and policy engineering and present a security policy life-cycle; an engineering methodology to policy development in high assurance computer systems. The model provides system security managers with a procedural engineering process to develop security policies. We also present an executable Prolog-based model as a formal specification and knowledge representation method using a theorem prover to verify system correctness with respect to security policies in their life-cycle stages.

Keywords: Logic, policy engineering, policy life-cycle, policy verification.

INTRODUCTION

High assurance computer systems are those that require convincing evidence that the system adequately addresses critical properties such as security, safety, and survivability[13]. They are used in environments where failure can cause security breaches or even the loss of life. Examples include avionics, weapons control, intelligence gathering, and life-support systems.

Security in high assurance computer systems involves protecting systems' entities from unauthorized (malicious or accidental) access to information. In this context, we use the following terms: entity to refer to any source or destination through which information can flow (e.g., user, subject, object, file, printer); security enclave (coalition) to refer to a logical boundary for a group of entities that have the same security level (e.g., CS faculty, ER physicians, C-130 pilots); and message to refer to any data that has been encoded for transmission to or received from an entity (e.g., a method invocation, a response to a request, a program, passing a variable, a network packet). The transmission mechanism can utilize shared memory, zero-copy message transport, kernel supported transport, TCP/IP, and so forth.

In this paper, we use the term policy to refer to security policy. In the computer security literature, the term policy has been used in a variety of ways. Policies can be a set of rules to manage resources (actions based on certain events) or definite goals that help determine present and future decisions. We provided a detailed discussion of the meaning of policy in high assurance computer systems in our earlier work[23]. Broadly speaking, a security policy shall address security issues: CIA (Confidentiality, Integrity, Availability). Confidentiality is related to the disclosure of information, integrity is related to the modification of information, and availability is related to the denial of access to information. The security policies discussed in this paper are multi-level (e.g., based on security classification: Top Secret, Secret, Confidential, Unclassified) and contain mandatory rules to guarantee that only authorized message transmission between entities can occur by imposing constraints on the actions (operations) of these entities. However, our work is not limited to military policies.

One fundamental key to successful implementation of secure high assurance computer systems is the design and implementation of security policies. These policies must specify the authorized transactions of the system and actions for unauthorized transactions, all in a form that is implementable. Implementing the enforcement of a policy is difficult and becomes very challenging when the system must enforce multiple policies. The purpose of a secure system is to provide configurations and mechanisms to support the functional use of system resources within the constraints of a specified policy. The specified security policy enumerates authorized access to information, or authorized information flow between components of the system.

Corresponding Author: Jim Alves-Foss, Director, Center for Secure and Dependable Systems, University of Idaho, P. O. Box 441008, Moscow, Idaho 83844-1008, USA. Tel.: +1-208-885-4114, Fax: +1-208-885-6840.

Our research focuses on the Multiple Independent Levels of Security (MILS) architecture, a high assurance computer system design for security and safety-critical multi-enclave systems. Although our research is not limited to MILS, it works well in this capacity. MILS is a joint research effort between academia, industry, and government led by the United States Air Force Research Laboratory with stakeholder input from many participants, including the Air Force, Army, Navy, National Security Agency, Boeing, Lockheed Martin, and the University of Idaho[1, 2, 12]. The MILS architecture is created to simplify the process of the specification, design, and analysis of high assurance computer systems[25]. This approach is based on the concept of separation, as introduced by Rushby[20].

The concept of separation is used, for example, in avionics systems and is a requirement of ARINC 653[4] (a standard for partitioning of computer resources) compliant systems. Through separation, we can develop a hierarchy of security services where each level uses the security services of a lower level or peer entities to provide a new security functionality that can be used by higher levels. Effectively, the operating system and middleware become partners with application level entities to enforce application-specific security policies. Limiting the scope and complexity of the security mechanisms provides us with manageable, and more importantly, evaluatable implementations. A MILS system isolates processes into partitions, which define a collection of data objects, code, and system resources. Partitions are defined by the kernel's configuration and can be evaluated separately. This divide-and-conquer approach will reduce the proof effort for secure systems.

We specified multi-policies in the MILS architecture using Inter-Enclave Multi-Policy (IEMP) and Policy Enforcement Graphs (PEG) in our earlier work[23, 24]. IEMP is an approach where all interactions between policies are controlled by a global multi-policy that guarantees the integration of several heterogeneous systems. For example, in a coalition model, IEMP can integrate Army, Air Force, and Navy forces with a joint staff that ensures policy-compliant interaction between the coalition members. PEG is a graph-based approach that creates policies that are used by IEMP providing a framework for supporting the enforcement of diverse security multi-policies. Although the approaches were designed for use in the MILS architecture, based on the concept of separation, they are applicable to a much broader range of architectures.

In this paper, we discuss the relationship between software engineering, security engineering, and policy engineering and present a policy development life-cycle; an approach to policy design in high assurance computer systems. We also present a Prolog-based policy formal specification and knowledge representation model. Prolog provides an implementation of a formal reasoning system, for example, it has the ability to make inferences. Formal inferences may be used for verifying the correctness of security policies and detecting conflicts between the policies, as well as supporting policy refinement.

POLICY ENGINEERING

We propose a security engineering methodology that provides system security managers with a procedural engineering process to develop security policies. Due to its essential role in bridging the gap between security requirements and implementation, policies should be taken into account early on in the development process.

System Engineering: To better understand the relationship between software engineering, security engineering, and policy engineering, we need to define each field and discuss how they are related to one another. We use the term system engineering to refer to the combination of software engineering, security engineering, and policy engineering. Software engineering is an approach that focuses on the tools and methods needed to develop cost-effective and faultless software systems[21]. Security engineering is a discipline that considers the development of systems that continue to function correctly under malicious attacks[3]. Security engineering focuses on the tools and methods needed to design, implement, and test dependable systems and must be integrated in every stage of the software development process. The goal of security engineering is to create confinement domains that will prevent malicious software from reading or modifying information[14]. Entities in the system cannot configure security policies; system security managers have the authority to evolve policies based on evolving requirements and management needs. System security managers ensure that entities in a system interact in a controlled way with one another and other systems. Systems should provide not only functionality, but also secure and safe end-to-end communication.

While policies in high assurance computer systems are considered strategies that establish a practical way to realize security models or requirements, they are often rules for logical deduction with the purpose of making decisions to protect resources from illegal usage[9]. Meanwhile, a set of policies (multi-policies) can be seen as a plan in the form of a program about

relationships between entities in a system. By executing the program, a system can automatically decide whether or not to permit access to its resources. Policy engineering is an approach similar to security engineering that can be applied in the generation and analysis of security policies.

Characterizing Policy Engineering: It is important to distinguish policy engineering research from research involving other engineering process areas. We consider policy engineering as a subfield of security engineering that distinguishes itself by being flexible; that is, the use of policies is separate from applications. This allows system security managers to build policies that can be specified, implemented, maintained, resolved when policy conflicts occur, and analyzed independently of application design (there is no need to modify applications in order to change policies). Policy engineering focuses on the development of systematic methods to design and enforce security policies. An engineering approach based on software engineering is applied to better design and implement security policies which will have an impact on the overall security of the system. While the software engineering approach ensures that an entity does a certain operation (e.g., entity A can write message m to entity B), policy engineering ensures that another entity does not (e.g., entity C cannot write message m to entity B).

Policy engineering ensures that the security enforcement mechanisms are NEAT (Non-bypassable, Evaluatable, Always invoked, and Tamperproof). Non-bypassable means that the mechanisms cannot be avoided even through the use of lower-level functions. Evaluatable means that the mechanisms are simple enough to be analyzed and mathematically verified. Always invoked means that the mechanisms are invoked every time an action occurs (they must mediate every access). Tamperproof means that the mechanisms cannot be changed by unauthorized entities.

Dai and Alves-Foss[9] stated the following similarities between software engineering and policy engineering:

- Multiple levels of development: abstract level for analysis, end user level for unskilled people writing policies, system level for policy refinement, and programmer level for policy coding and integration.
- Multiple representation languages: mathematical symbols for formal specification and analysis, visual language for unskilled people writing policies, programming language for implementing and enforcing policies.
- Multiple validation approaches: formal verification and testing can be applied to validate policies.

However, policies distinguish themselves from system or application software of other functionalities by their logical nature. Policy enforcement is a logical deduction process and specified policies may logically contradict one another. Consequently, conflict detection is important during policy development.

Dai and Alves-Foss[9] argued that in policy engineering, policies are separated away from the system and application software of other functionalities and treated specifically. They stated that the rationale behind this strategy includes the following:

- Convenience for security analysis: security analysis of a system can be separated into two parts: interface implementation analysis and policy analysis. The interface primarily refers to the policy enforcement mechanism that may be provided as an application programming interface and integrated into other functionalities of a program. Once the interfaces have been proved to be correctly and safely established, security analyzers only need to focus on the validation of policies.
- Convenience for coding: software developers can focus on implementing non-security related algorithms and embedding the general policy enforcement mechanism in their software. This decreases the complexity of the code, as programmers do not have to entwine user policies in their software.
- Scalability of system security: system and end users have the capability to flexibly create, modify, delete, and analyze their security policies for a system or an application based on their own needs.

Consider the policy engineering architecture depicted in Fig. 1. We believe that in order to have systems that are dependable, with cost-effective software, and with flexible policies that offer a management mechanism, all engineering approaches (software engineering, security engineering, policy engineering) are needed in each step of creating and maintaining high assurance computer systems.

We believe that policy engineering has been ignored and has not been included in the development process due to several reasons, including:

- Most system security managers, although acknowledge the importance of policy design, lack the knowledge and experience of policy development methods.

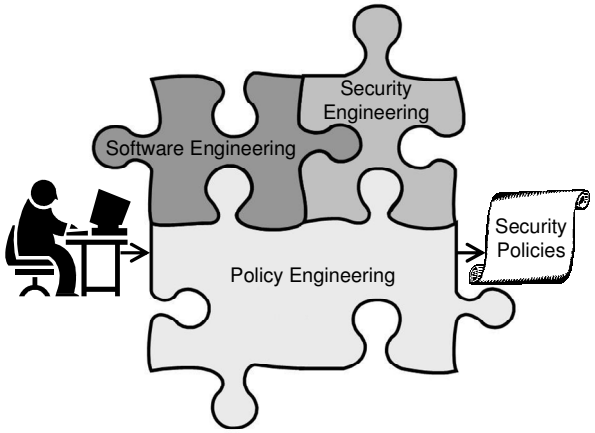

Fig. 1: Policy engineering architecture.

- Most system security managers are not willing to sacrifice security with marketing of the product; they are under pressure to get the product shipped to consumers.
- Analyzing security policy behavior for consistency and completeness can be a difficult task.

POLICY LIFE-CYCLE

High-level policies are statements that define objectives or entity relationships. They are not specific to the deployed technology (e.g., which communication ports, protocols, and devices are used). The following are examples of high-level policies: "Allow only detectives to access the evidence room", "A faculty member can view university records only for students who are currently enrolled in his or her course". High-level policies are refined into low-level policies that a computer can interpret. Low-level policies are specific to the deployed technology. For example, depending on the network configuration, communication port, protocol, and device, policies must be identified in a specific format. Low-level policies result from mapping of high-level policies in a specific environment. The following is an example of a low-level policy: "IF SourceIPAddress = 1.1.1.1 AND SourcePort = 200 AND DestinationIPAddress = 7.7.7.7 AND DestinationPort = 500 AND Protocol = GIOP AND Dominates(SourceSecurityClassification, DestinationSecurityClassification) THEN AccessAllowed(Source, Destination) ELSE AccessDenied(Source, Destination)".

Security should be formally integrated into the development life-cycle. The policy development life-cycle is the process of developing security policies. The proposed model describes the series of ongoing steps through which a policy progresses and the order in which those steps must be followed by system security managers. In this structured model, the process is divided into five stages. During each stage, distinct activities take place with their own input, output, and analysis techniques. Each stage uses documentation of a previous stage to accomplish its objectives.

Stages: Policies go through various development stages throughout their life before they are deployed. Figure 2 shows the five stages of the policy life-cycle model: requirements, design, implementation, enforcement, and enhancement (RDIEE).

1. Policy requirements analysis: during this stage, business needs and constraints are identified. Descriptions of entities, enclaves, and security policy goals (high-level) are provided. A policy requirements document is generated, which will be a guide for the policy designers.
2. Policy design: during this stage, the policy requirements document is reviewed by the policy designers who design techniques by which the requirements will be implemented. High-level policies are refined and transformed into low-level policies, algorithms are designed, and the following are identified: modules, interfaces, threats, and risks. During this stage, a policy design document is generated that specifies what the system does.
3. Policy implementation: during this stage, implementation begins with the policy design document and produces code in a programming language. Entity predicates and interfaces are coded and the code is tested in a stand-alone environment (one domain, not communicating with inter-domain guards or network communication devices) to identify any potential errors.
4. Policy enforcement: during this stage, inter-domain guards and network communication devices (e.g., encryption engine, message router, wireless access point, downgrader) are used to mediate message passing between the system's entities. Low-level policies are enforced by the enforcement mechanisms (the mechanisms are not specific to one policy; they ensure that the refined policies are enforced by the right entities) and policies are monitored to ensure that the system as a whole correctly implements policies. The behavior of the system and its components are verified. At the end of this stage, a working system is produced that has been tested and is compliant with policy requirements.

5. Policy enhancement: during this stage, the system evolves to meet any changes in policy requirements. Several issues are addressed to ensure that the system as a whole still meets policy requirements: do policies meet the requirements, do policies operate effectively, detect and resolve policy conflict or defect, audits (records that provide a documentary of actions which is often used to trace a system, audits will be performed for each entity request in the system; a request will be logged in as a trace operation which will be used for analysis of activities in the system), and policy assurance; that is, security requirements in the policy are consistent and complete. They are consistent because an entity request is either accepted or denied but not both and complete because for each entity request, there is a result (the access being accepted or denied).

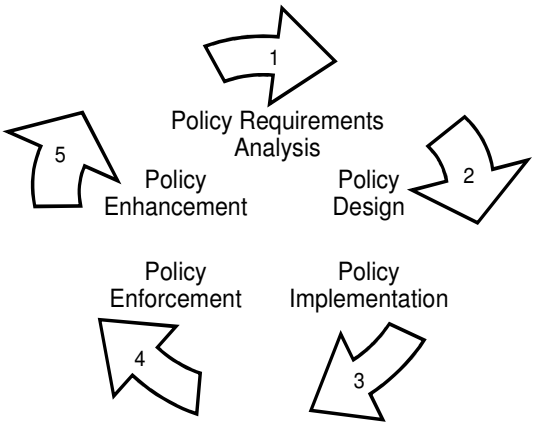

Fig. 2: Policy life-cycle model.

Example: The following example illustrates some activities that take place during each stage of the RDIEE life-cycle process:

1. Requirements: consider the problem of designing a program that facilitates granting or denying entity access. A request is made by a source entity attempting to access a destination entity. Given the source entity, source enclave, source role, operation (read or write), destination entity, destination enclave, and destination role, the program should respond with a decision: access is either granted or denied. The system access policy is based on the simple security and star properties as defined by Bell-LaPadula[5] (no read-up and no write-down, respectively). There are five entities, four enclaves, five roles, and two operations. The program must be completed within three months with a cost of less than $4,000.
2. Design: define the system's entities, enclaves, roles, and operations. Assign enclaves, roles, and security classifications (1 = Unclassified, 2 = Confidential, 3 = Secret, 4 = Top Secret) to the entities. Define a dominate() predicate that is based on the simple security and star properties as defined by Bell-LaPadula[5]. Users will input the following information: source entity name, the enclave that it belongs to, its role, operation, destination entity name, the enclave that it belongs to, and its role; all variables are of data type atom. In addition, define allow() predicates that process read operations for entities that are members of one enclave (the same enclave), write operations for entities that are members of one enclave, read operations for entities that are members of different enclaves, and write operations for entities that are members of different enclaves. The allow() predicates will be invoked to determine whether access is granted or denied. If access is granted, then the program displays "yes" and if the access is denied, then the program displays "no". Users can also have queries about the system's entities, enclaves, roles, and operations.
3. Implementation: the modules of the system design are coded. See the Prolog source code in the Implementation Section.
4. Enforcement: the program is validated and tested for correctness (see the Prolog Section). Sample test cases (valid and invalid) are provided that demonstrate a working system that meets policy specifications. In addition to testing the program against test cases, a code review (inspection) is done to detect and resolve any programming faults. The reviewers ensure that there are no contradictions and ambiguities between the policy requirements and policy design activities.
5. Enhancement: the following entities are added to Prolog's knowledge base: Raneem, Dana, and Sam and a new role, called post_doc, is defined. Enclaves, roles, and security classification are assigned to the added entities as shown in the Enhancement Section. The program is tested to ensure that not only the policy changes have been correctly implemented, but also that the functionality of the rest of the program has not been contradicted.

Advantages and Limitations: One of the key advantages of the RDIEE policy life-cycle model is that it allows system security managers to have control over

each clearly-stated individual development stage, which promotes manageability. Another advantage is that evolving system characteristics can be incorporated in the model. A limitation of the model is that a policy change requires time to implement in the system due to the fact that policies may affect one another and careful modification is required. Another limitation is that the whole system is tested towards the end of the process.

PROLOG

In order to verify security policies in their life-cycle (stages 3, 4, and 5), we use Prolog as a method to prove system correctness with respect to policies using Prolog's theorem prover that is based on a special strategy for resolution called SLD-resolution. Prolog is a first-order predicate logic programming language that uses Horn clauses to describe relationships based on mathematical logic. A Prolog program consists of clauses stating facts and rules. Facts and rules are explicitly defined and implicit knowledge can be extracted from Prolog's knowledge base. Queries are used to check whether relationships hold.

Unlike programs in other programming languages, a Prolog program is not a sequence of processing steps to be executed. A Prolog program is a set of formulas (axioms) from which other formulas expressing properties of this program may be deduced as theorems[17]. A theorem is proved using a proof procedure, that is a sequence of rules of inference producing new expressions from old ones, the inference rules are repeatedly applied to a set of axioms and the new expressions, until the desired theorem is reached[8]. The theorem to be proved is the starting goal (clause) and the inference rules and axioms are the program itself.

Benefits: Prolog has several benefits that are appealing, including:

- Compliance: code can be written to meet security and safety needs.
- Flexibility: information access can be defined and controlled.
- Mathematical logic: Prolog's syntax and meaning can be concisely specified with reference to logic.
- Dynamic nature: information and constraints can be added to the existing knowledge base while the program is executed.
- Efficiency: a small number of lines of code can be written to perform a task.
- Interoperability: Prolog can communicate with different applications that use other programming languages and software visualization tools.

Because of its simple declarative nature, Prolog is an appropriate language for expressing and verifying security policies. To determine all possible answers to a query, Prolog supports backtracking. Backtracking is the process of determining all facts or rules with which unification (determining whether there is a substitution that makes two atoms the same) can succeed. Several authors in the literature indicated the use of Prolog as a policy specification language, including Lin[16] who argued that Prolog is a suitable language for specifying security policies due to several features, among which is that it is based on a subset of first-order logic with a solid mathematical foundation, it is a productive modeling language supporting incremental policy writing and refinement, it is able to reason from a set of rules, and it supports meta-level reasoning thus making policy conflict detection possible.

Implementation: The following is an excerpt from a Prolog-based implementation model that we have developed. The expressed policy is based on the simple security and star properties as defined by Bell-LaPadula[5]. As demonstrated in the model, we use a closed security policy where only the allowed operations are specified; only authorized entities are allowed access. The operations to be denied are not explicitly specified because Prolog's negation-by-failure mechanism will enforce a default denial on messages other than those explicitly allowed by the knowledge base and inference rules.

Entities, enclaves, and roles in the system are defined as Prolog facts. The sets Entity = {Penny, Diala, Adrian, Trudy, Evey}, Enclave = {1, 2, 3, 4}, and Role = {Faculty, Staff, BS Student, MS Student, PhD Student} are defined as follows:

```
entity(penny).
entity(diala).
entity(adrian).
entity(trudy).
entity(evey).

enclave(enc1).
enclave(enc2).
enclave(enc3).
enclave(enc4).

role(faculty).
role(staff).
role(bs_stud).
role(ms_stud).
role(phd_stud).
```

The allowed operations between entities are read or write, defined as Prolog facts as follows:

```
op(read).
op(write).
```

Enclaves, roles, and security classifications, which are defined as Prolog facts, are assigned to entities. The format of the classification() predicate is classification(entity, enclave, role, security_classification). We use the following values as follows: 1 to refer to a security classification of Unclassified, 2 to refer to a security classification of Confidential, 3 to refer to a security classification of Secret, and 4 to refer to a security classification of Top Secret. The facts are defined as follows:

```
classification(penny,enc1,faculty,4).
classification(diala,enc1,faculty,4).
classification(adrian,enc4,staff,1).
classification(trudy,enc3,ms_stud,2).
classification(evey,enc1,phd_stud,4).
classification(evey,enc2,bs_stud,3).
```

After Prolog facts have been defined, Prolog rules are stated as follows:

- Rule 1 processes only read operation requests for entities that are members of one enclave (the same enclave). The format of the allow() predicate is allow(EntityA, EnclaveA, RoleA, read, EntityB, EnclaveB, RoleB). The rule is defined as follows:

```
allow(EntA,EncA,RoleA,Oper,EntB,EncB,RoleB):-
   op(Oper),
   Oper = read,
   EncA = EncB,
   role(RoleA),
   role(RoleB),
   entity(EntA),
   entity(EntB),
   enclave(EncA),
   enclave(EncB),
   !.
```

- Rule 2 processes only write operation requests for entities that are members of one enclave. The format of the allow() predicate is allow(EntityA, EnclaveA, RoleA, write, EntityB, EnclaveB, RoleB). The rule is defined as follows:

```
allow(EntA,EncA,RoleA,Oper,EntB,EncB,RoleB):-
   op(Oper),
   Oper = write,
   EncA = EncB,
   role(RoleA),
   role(RoleB),
   entity(EntA),
   entity(EntB),
   enclave(EncA),
   enclave(EncB),
   !.
```

- Rule 3 processes only read operation requests that are members of different enclaves. The format of the allow() predicate is allow(EntityA, EnclaveA, RoleA, read, EntityB, EnclaveB, RoleB). The rule is defined as follows:

```
allow(EntA,EncA,RoleA,Oper,EntB,EncB,RoleB):-
   op(Oper),
   Oper = read,
   role(RoleA),
   role(RoleB),
   entity(EntA),
   entity(EntB),
   enclave(EncA),
   enclave(EncB),
   not_equal(EncA,EncB),
   dominate(EntA,EncA,RoleA,EntB,EncB,RoleB),
   !.
```

- Rule 4 processes only write operation requests that are members of different enclaves. The format of the allow() predicate is allow(EntityA, EnclaveA, RoleA, write, EntityB, EnclaveB, RoleB). The rule is defined as follows:

```
allow(EntA,EncA,RoleA,Oper,EntB,EncB,RoleB):-
   op(Oper),
   Oper = write,
   role(RoleA),
   role(RoleB),
   entity(EntA),
   entity(EntB),
   enclave(EncA),
   enclave(EncB),
   not_equal(EncA,EncB),
   dominate(EntB,EncB,RoleB,EntA,EncA,RoleA),
   !.
```

- Rule 5 determines whether entities dominate one another (based on the simple security and star properties as defined by Bell-LaPadula[5]). The format of the dominate() predicate is dominate(EntityA, EnclaveA, RoleA, EntityB, EnclaveB, RoleB). The rule is defined as follows:

```
dominate(EntA,EncA,RoleA,EntB,EncB,RoleB):-
   classification(EntA,EncA,RoleA,CL1),
   classification(EntB,EncB,RoleB,CL2),
   CL1 >= CL2.
```

- Rule 6 consists of two sub-rules: not_equal() and not() predicates that determine whether two identifiers are equal. The sub-rules are defined as follows:

```
not_equal(X,Y):-
   not(X = Y).
```

```
not(X):-
   X, !, fail
   ;
   true.
```

Enforcement: In order to test the Prolog program for correctness, valid and invalid test cases are provided. The test cases demonstrate a working system that meets policy specifications. The following are sample queries:

- Can faculty Penny in enclave 1 read from staff Adrian in enclave 4?
  ```
  | ?- allow(penny,enc1,faculty,read,adrian,enc4,staff).
  yes
  ```
- Can BS student Evey in enclave 2 read from PhD student Evey in enclave 1?
  ```
  | ?- allow(evey,enc2,bs_stud,read,evey,enc1,phd_stud).
  no
  ```
- Can MS student Trudy in enclave 3 write to faculty Penny in enclave 1?
  ```
  | ?- allow(trudy,enc3,ms_stud,write,penny,enc1,faculty).
  yes
  ```
- Can faculty Diala in enclave 1 write to staff Adrian in enclave 4?
  ```
  | ?- allow(diala,enc1,faculty,write,adrian,enc4,staff).
  no
  ```
- Can MS student Trudy in enclave 3 read from staff Sam in enclave 5?
  ```
  | ?- allow(trudy,enc3,ms_stud,read,sam,enc5,staff).
  no
  ```
- Who are the members of enclave 1?
  ```
  | ?- classification(Entity,enc1,Role,4).
  Entity = penny
  Role = faculty ? ;

  Entity = diala
  Role = faculty ? ;

  Entity = evey
  Role = phd_stud ? ;
  ```
- Give all entities with their assigned enclaves, roles, and security classifications.
  ```
  | ?- classification(Ent,Enc,Role,SecCl).
  Enc = enc1
  Ent = penny
  Role = faculty
  SecCl = 4 ? ;

  Enc = enc1
  Ent = diala
  Role = faculty
  SecCl = 4 ? ;

  Enc = enc4
  Ent = adrian
  Role = staff
  SecCl = 1 ? ;

  Enc = enc3
  Ent = trudy
  Role = ms_stud
  SecCl = 2 ? ;

  Enc = enc1
  Ent = evey
  Role = phd_stud
  SecCl = 4 ? ;

  Enc = enc2
  Ent = evey
  Role = bs_stud
  SecCl = 3 ? ;
  ```
- Is system administrator a role?
  ```
  | ?- role(sys_admin).
  no
  ```
- Can faculty Ray in enclave 1 read from staff Adrian in enclave 4?
  ```
  | ?- allow(ray,enc1,faculty,read,adrian,enc4,staff).
  no
  ```
- Can MS student Trudy in enclave 3 execute faculty Penny in enclave 1?
  ```
  | ?- allow(trudy,enc3,ms_stud,execute,penny,enc1,faculty).
  no
  ```

Enhancement: The Prolog program is tested to ensure that the policy changes have been correctly implemented and that the functionality of the rest of the program has not been contradicted.

- New entities Raneem, Dana, and Sam are added to Prolog's knowledge base as follows:
  ```
  entity(raneem).
  entity(dana).
  entity(sam).
  ```
- A new role, called post_doc, is defined as follows:
  ```
  role(post_doc).
  ```
- Enclaves, roles, and security classification are assigned to the added entities:
  - Raneem is assigned to enclave 1, a role of post_doc, and a security classification of 4 as follows:
    ```
    classification(raneem,enc1,post_doc,4).
    ```
  - Dana is assigned to enclave 3, a role of faculty, and a security classification of 2 as follows:
    ```
    classification(dana,enc3,faculty,2).
    ```
  - Sam is assigned to enclave 4, a role of staff, and a security classification of 1 as follows:
    ```
    classification(sam,enc4,staff,1).
    ```
- An existing entity Penny (who is in enclave 1) is assigned to additional enclaves, enclaves 2 and 3 as follows:
  ```
  classification(penny,enc2,faculty,3).
  ```

```
classification(penny,enc3,faculty,2).
```

After the changes were made, the program was executed. The results confirm that the policy changes have been correctly implemented and that the rest of the program has not been compromised (the new results are the same as those obtained before the changes were made).

In the next four paragraphs, we summarize related research work regarding policy life-cycle models, policy engineering, security engineering, and policy specification in Prolog.

We found very little research in the literature that considers a life-cycle for security policies. Among those we did find was the model defined by Goh[11] who presented a policy evolution life-cycle that consists of the following stages: requirements establishment, domain interpretation, iterative refinements and object specifications, configuration mapping, and enforcement verification. More recently, Cakic[7] presented a policy life-cycle model using a state transition diagram. The model consists of the following main phases: specification, enabling, enforcement, modifications, and deletion.

Shah et al.[22] introduced consistency maintenance between different policies within a given domain or policies across domains using policy engineering. They argued that consistency maintenance for policies across domains is required when several domain ontologies are merged together and policies are being created on entities within these domains. Lewis et al.[15] proposed a method to support policy engineering by using ontology-based semantic models of the managed system to enable automated reasoning about the resolution and interactions of policies.

Irvine[14] indicated the need for an early step in the security engineering process: developing a mathematical security policy model. The formal model should demonstrate that the policy is not flawed and that the operations described in the system do not violate the security policy. Bryce[6] presented a design for a security system that provides a security engineering framework for the design, verification, and implementation of application security policies.

Although various languages have been proposed for specifying policies for different purposes, a standard language does not yet exist for the policy community to use. Moffett and Sloman[19] provided an analysis of policy hierarchies by specifying policy hierarchy refinement relationships in Prolog. Lupu and Sloman[18] applied Prolog to meta-policies to identify several types of policy conflicts. DeTreville[10] presented Binder, a logic-based security language that provides low-level programming tools to implement security policies. Binder adopted Prolog's syntax and its programs can be translated into Prolog.

CONCLUSIONS

This paper outlines a policy engineering methodology that provides system security managers with a procedural process to develop security policies in high assurance computer systems. We propose a policy life-cycle model that is significant in high assurance computer systems because policies are crucial elements of systems' security (i.e., defining a policy life-cycle model will lead to having more secure systems). Before policies can be deployed, it is essential that the development life-cycle starts with a clearly-stated policy requirements analysis and goes through policy design, policy implementation, policy enforcement, and finally policy enhancement. Policies are usually designed not only to guide information access, but also to control conflicts and cooperation of security policies of different security enclaves. We strongly believe that no enforcement of security standards can be effectively made without the support of security policies.

Integrating security is a vital issue in computer systems and software engineering. Systems' security should be considered before design. An engineering approach to policy forms the foundation to security. Therefore, it is crucial for system security managers to constantly keep a policy's effect in mind throughout a system's development life-cycle. Further work is needed to better understand the impact of security policy development on effective security design. The relationship between software engineering, security engineering, and policy engineering introduces new challenges that need to be investigated. The approach proposed in this paper is an important step towards defining this relationship.

ACKNOWLEDGEMENTS

We wish to acknowledge the United States Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) for their support. This material is based on research sponsored by AFRL and DARPA under agreement number F30602-02-1-0178. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be

interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, or the U.S. Government. We also wish to acknowledge the anonymous reviewers and journal editors for reviewing this paper.

REFERENCES

1. Alves-Foss, J., W. S. Harrison, P. Oman, and C. Taylor, 2006. The MILS architecture for high assurance embedded systems. International Journal of Embedded Systems, 2 (3/4): 239-247.
2. Alves-Foss, J., C. Taylor, and P. Oman, 2004. A multi-layered approach to security in high assurance systems. In Proceedings of the 37th Annual Hawaii International Conference on System Sciences.
3. Anderson, R., 2001. Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley & Sons.
4. Avionic application software standard interface (Draft 3 of Supplement 1) (Specification ARINC 653), 2003. ARINC Standards.
5. Bell, D. E. and L. J. LaPadula, 1976. Secure computer systems: Unified exposition and MULTICS interpretation. Technical Report ESD-TR-75-306, MITRE Corporation MTR-2997 Rev. 1.
6. Bryce, C., 1997. The Skippy security engineering framework. Technical Report 1060, GMD - German National Research Centre for Information Technology.
7. Cakic, J., 2003. A High-Level Framework for Policy-Based Management of Distributed Systems. PhD thesis, University of Kent at Canterbury.
8. Civera, P., G. Masera, G. Piccinini, and M. Zamboni, 1994. VLSI Prolog Processor, Design and Methodology: A Case Study in High Level Language Processor Design. North-Holland.
9. Dai, J. and J. Alves-Foss, 2002. Logic based authorization policy engineering. In Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics, pp: 230-238.
10. DeTreville, J., 2002. Binder, a logic-based security language. In Proceedings of the IEEE Symposium on Security and Privacy, pp: 105-113.
11. Goh, C., 1998. Policy management requirements. Technical Report HPL-98-64, Hewlett-Packard Laboratories.
12. Harrison, W. S., N. Hanebutte, P. Oman, and J. Alves-Foss, 2005. The MILS architecture for a secure global information grid. Crosstalk: The Journal of Defense Software Engineering, 18 (10): 20-24.
13. Heitmeyer, C., 2004. Managing complexity in software development with formally based tools. Electronic Notes in Theoretical Computer Science, 108: 11-19.
14. Irvine, C. E., 2000. Security: Where testing fails. International Test and Evaluation Association Journal, 21 (2): 53-57.
15. Lewis, D., K. Feeney, K. Carey, T. Tiropanis, and S. Courtenage, 2004. Semantic-based policy engineering for autonomic systems. In Proceedings of the 1st IFIP TC6 WG6.6 International Workshop on Autonomic Communication, pp: 152-164.
16. Lin, A., 1999. Integrating policy-driven role based access control with the common data security architecture. Technical Report HPL-1999-59, Hewlett-Packard Laboratories.
17. Loeckx, J. and K. Sieber, 1987. The Foundations of Program Verification. John Wiley & Sons and B. G. Teubner, 2nd edition.
18. Lupu, E. C. and M. Sloman, 1999. Conflicts in policy-based distributed systems management. IEEE Transactions on Software Engineering, 25 (6): 852-869.
19. Moffett, J. D. and M. S. Sloman, 1993. Policy hierarchies for distributed systems management. IEEE Journal on Selected Areas in Communications, 11 (9): 1404-1414.
20. Rushby, J. M., 1981. Design and verification of secure systems. In Proceedings of the 8th ACM Symposium on Operating System Principles, pp: 12-21.
21. Schach, S. R., 2005. Object-Oriented and Classical Software Engineering. McGraw-Hill, 6th edition.
22. Shah, A., L. Kagal, T. Finin, and A. Joshi, 2004. Policy development software for security policies. In Proceedings of the DIMACS Workshop on Usable Privacy and Security Software.
23. Wahsheh, L. A. and J. Alves-Foss, 2006. Specifying and enforcing a multi-policy paradigm for high assurance multi-enclave systems. Journal of High Speed Networks, 15 (3): 315-327.
24. Wahsheh, L. A. and J. Alves-Foss, 2007. Using policy enforcement graphs in a separation-based high assurance architecture. In Proceedings of the IEEE International Conference on Information Reuse and Integration, pp: 183-189.
25. White, P., W. Vanfleet, and C. Dailey, 2000. High assurance architecture via separation kernel. Draft.