Implementation-Oriented Secure Architectures

Daniel Conte de Leon, Jim Alves-Foss, and Paul W. Oman
Center for Secure and Dependable Systems
University of Idaho
[danielc, jimaf, oman] at cs.uidaho.edu

Abstract

We propose a framework for constructing secure systems at the architectural level. This framework is composed of an implementation-oriented formalization of a system's architecture, which we call the formal implementation model, along with a method for the construction of a system based on elementary analysis, implementation, and synthesis steps. Using this framework, security vulnerabilities can be avoided by constraining the architecture of a system to those architectures that can be rigorously argued to implement all corresponding functional and security requirements, and no other. Furthermore, the framework enables the verification and validation of system correctness by enforcing traceability of final system components to their corresponding design, architecture, and requirement work products.

1. Introduction

High assurance and critical computing systems require compelling evidence that the final developed system satisfies a defined set of critical properties as well as a defined set of functional requirements. Both nonfunctional requirements, also called system constraints, and functional requirements contribute to system correctness; a system is correct if it correctly, completely, and uniquely implements every functional requirement while at the same time it correctly, completely, and uniquely implements every nonfunctional requirement [5].

Nonfunctional requirements are referred to as dependability properties [1]. Dependability properties of a high assurance and critical computing system such as security and safety are *emergent* properties of the system [17].

The Stanford Encyclopedia of Philosophy offers a good introduction to the concepts of emergence and emergent properties [25]. "Emergent properties are systemic features of complex systems which could not be predicted (practically speaking; or for any finite knower; or for even an ideal knower) from the standpoint of a pre-emergent stage, despite a thorough knowledge of the features of, and laws governing, their parts" [25]. Emergence is intrinsically related to the notion of levels of abstraction: an emergent property "emerges" at a higher level of abstraction but it cannot be observed at any of the previous lower levels of abstraction [6].

Complex systems are built as a set of interacting components. A system model describes and/or specifies those interacting components and their relationships at different levels of abstraction. In order to anticipate and verify the emergent properties of a system by using a system model, it is necessary to position ourselves at a higher level of abstraction from that of the components being analyzed in the system model. Hinton exemplified how undesired or unexpected emergent properties can arise at higher levels of abstraction (i.e., at the system level) by showing how an insecure system can be composed of two Generalized Non-Interference (GNI) secure components [9], and Leveson showed how unexpected emergent behaviors resulted in accidents in safety-critical systems [18].

Current state of the practice approaches to systems and software requirements, architecture, and implementation do not facilitate the analysis of emergent security properties. This is due to the fact that most current approaches do not enforce the creation and maintenance of essential traceability links which establish the relationship between system components and their, design, architecture, and functional and nonfunctional requirement counterparts. For example, most current approaches are focused on a divide and conquer strategy, which is necessary for complex systems development, but they do not facilitate the analysis of emergent security properties that appear at higher levels of abstraction after system components are built, integrated, and deployed.

To solve this persistent problem, we argue that hybrid approaches to system specification and development are necessary. We propose a framework for the incremental development of secure systems that is formal but, at the same time, allows the level of formalization to evolve along with system analysis, design, and implementation. In addition, this framework enforces a strong connection, by the

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Implementation-Oriented Secure Architectures	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Idaho,Center for Secure and Dependable Systems,PO Box 441008,Moscow,ID,83844-1008**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**10**	

use of formal traceability relations between all work products and at all levels of abstraction. By using the proposed framework, security vulnerabilities can be avoided by constraining the architecture of a system to those architectures that can be shown to completely and uniquely implement all corresponding functional and nonfunctional security requirements, including environmental constraints.

The proposed framework is composed of:

I. **A Formal and Implementation-Oriented Architectural Model of a System:** In this framework the architecture of a system is represented using a **formal implementation model** that takes into account all deliverables resulting from system analysis, development, and deployment processes.

II. **A Method for Systems Development:** A method for the incremental development of systems that comply with a set of implementation constraints over a formal implementation model (implementation-oriented architectural model). In a formal implementation model the level of formalization may incrementally evolve along with the analysis, design, and implementation processes.

The remainder of this paper is organized as follows: Section 2 justifies our hybrid (formal and informal) approach to systems development. Section 3 briefly describes the formal foundation of this framework, which is based on previous work by the authors which focused on the discovery of hidden dependencies in order to anticipate safety hazards in safety-critical systems [4, 5]. Sections 4 and 5 describe the framework for the development of secure system architectures. To demonstrate how this proposed framework would help build more secure systems, in Section 6 we show how two security vulnerabilities could have been avoided by using the proposed framework. Our conclusions along with some open questions are presented in Section 7. Lastly, Section 8 briefly reviews related work.

2. Merging Formal and Informal System Development Approaches

A set of well-known engineering and design methodologies is used today for the analysis of high assurance and critical computing systems and their separation into constructible or realizable components. Furthermore, there exists a set of methods for the creation and later synthesis of those developed components. For example, requirements engineering methods are used to elicit system requirements and to further divide those requirements into derived (i.e., child) requirements [11], and architectural analysis and specification methods are used to develop and specify the architecture of a system including hardware and software [2]. In addition, build utilities, compilers, linkers, and assemblers are used to construct the software of a system based on the source code, and hardware component assembly and connection techniques are used to assemble system's hardware.

However, methods for the formal analysis and verification of security properties of a final assembled or in-production system are scarce and not commonly used in industry. While it has been demonstrated that formal approaches for developing secure systems can help to foresee major flaws in the design of a system, most formalizations are disconnected with the initial informal requirements and typically also disconnected with the final implemented system as well. In other words, except for formal development systems like Kestrel's Specware®[1] approach [12, 13], the Galois Haskell-based approach [7, 14], and the Praxis High Integrity Systems SPARK Ada approach [8, 19] there is rarely formal connection between the formal models and the final system implementation. Furthermore, there have been several cases where formalizations of security protocols have been found semantically flawed, mainly due to the fact that assumptions made during the formalization process did not hold true in the final implemented system or that the system was underspecified in its formal representation (i.e., missing assumptions) [9].

Purely formal approaches have been shown to be very useful for developing dependable systems, but they also carry burdens such as lack of trained personnel and large learning curves that have precluded widespread adoption. There is also, for most formal methodologies, a lack of a strongly structured (or formal) connection with both ends of the system development process: informal requirement descriptions and final (in-production) system implementations.

On the other hand, standard programming methods as currently used by industry, including object-oriented programming languages, have also failed to create secure systems mainly due to their inability to enable the verification of systemic (emergent) properties or nonfunctional requirements and their disconnection with system requirements and other formal and informal system descriptions or specifications.

We do not argue in this paper for or against either formal or informal system and software development methods, we recognize the value of both. However, we point out that when used separately and without strongly structured connections between them these methods have failed to achieve the promised goals. New dependable system development methods that combine formal and informal approaches are necessary.

The proposed framework reuses some aspects of Leveson's *Intent Specifications* and *SpecTRM* methodologies, which have proved successful for the specification of safe

[1] Specware® is a registered trademark of the Kestrel Institute.

air traffic control and airborne systems [15, 16] (see the Related Work section). It also extends our previous work, which is briefly described in the next section.

3. Formal Implementation Models

A formalization of traceability relationships based on set theory and predicate calculus was initially introduced in Conte de Leon [3]. An extended analysis of this formalization for the specific case of the partial implementation relationship and an improved description of the formal system appears in Conte de Leon and Alves-Foss [5].

In our work, we divide system components, architectural and design artifacts, and requirements into what we call **work product sections**, which we define as uniquely identifiable, addressable, and meaningful sections of work products [22]. We abbreviate work product sections as wps and wps(s) for singular and plural, respectively. We call $\mathbb{S}$ the set of all wps(s) of a system. We call *implements* the partial order established by the partial implementation traceability relation between wps(s) in $\mathbb{S}$. Further, we denote the top and bottom of our abstract formal model as $\top$ and $\bot$, respectively.

The formal system Γ described by Conte de Leon and Alves-Foss is based on set theory and a binary predicate calculus. In Γ, each traceability link between any two wps(s) is formalized as an axiom or theorem. In the same article, we analyzed and formalized the structural properties of the partial implementation traceability relationship between wps(s). We formalized the partial implementation traceability relationship as a strict partial order (irreflexive, asymmetric, and transitive). This strict partial order between wps(s) along with a new technique named *conceptual completeness* were applied in order to discover hidden dependencies between wps(s) that led to safety hazards which developed into catastrophic accidents. A prototype expert system (SyModEx) was constructed to verify and enforce these properties, among others, as inference rules of Γ.

In this paper, we extend the Γ formal system introduced in Conte de Leon and Alves-Foss [5] with a different objective: we create new inference rules that constrain the evolution of a formal implementation model of a system in order to avoid security vulnerabilities. These rules enforce the method for secure systems development explained in Section 5 of this paper. However, these rules do not specify the processes, methodologies, or languages used to create these architectures, they only constrain the architecture of a system to a set of "safe" architectural states.

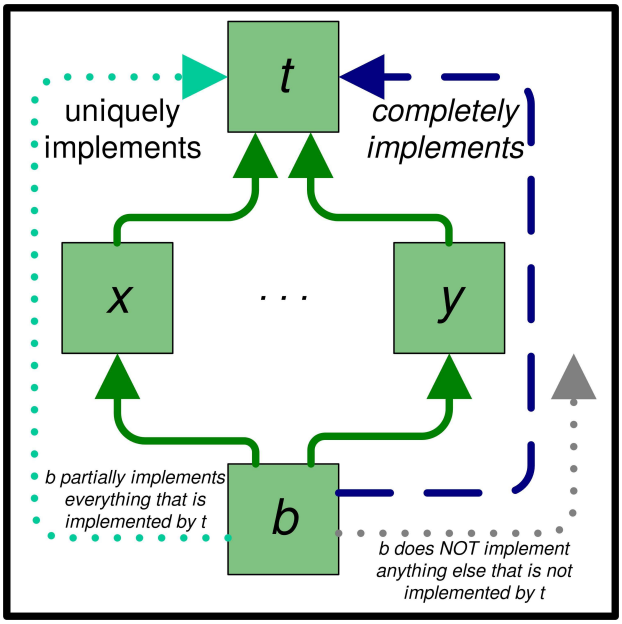

Figure 1. Complete and Unique Implementation of Work Product Sections.

4. Framework Definitions: Partial, Complete, and Total Implementation of Work Product Sections

In this paper, we augment the formal model Γ with three traceability relationship types and three inference rules. The three new relationship types are: *completelyImplements*, *uniqelyImplements*, and *totallyImplements*. We define and describe these newly introduced concepts below.

Definition 1. ***Complete implementation between work product sections:*** *A wps b (implementor)* ***completely implements*** *a wps t (implementee) if and only if every chain $\langle C, implements \rangle$ in Γ, whose minimum element is $\bot$ and whose maximum element is t contains b.*

This definition can also be stated as: a wps b (implementor) completely implements a wps t (implementee) if and only if every intermediate wps that is partially implemented by b also partially implements t. Also, this is equivalent to say that every directed path from $\bot$ to t contains b. Figure 1 illustrates this definition with a diagram.

Informally, a wps b **uniquely implements** another wps t if and only if b does not directly or indirectly partially implement a third wps. In other words there are no partial implementation traceability links going out of b that do not go through t. In this, case we could also say that t is uniquely implemented by b.

Definition 2. ***Unique implementation between work product sections:*** *An implementor wps b* ***uniquely implements*** *an implementee wps t if and only if every chain $\langle C, implements \rangle$ in Γ, whose maximum element is $\top$ and whose minimum element is b contains t.*

Definition 3. ***Total Implementation between two work product sections:*** *A wps b (implementor) totally implements a wps t (implementee) if and only if b completely and uniquely implements t.*

If b totally implements t then there is a *lattice-like* algebraic structure embedded in Γ, with respect to the implements traceability relation, such that b is the $\bot$ and t is the $\top$. In this structure, given any pair (x, y) of wps(s), the set of lower bounds and the set of upper bounds of (x, y) are nonempty. Using this algebraic structure we can mathematically verify that a given formal implementation model, and therefore the architecture of a given system, complies with the set of rules defined in Section 5. As we will see in the next section, however, the formal model is not a proper lattice it is only lattice-like.

5. System Development Method

Our proposed system development method is composed of six steps:

1. basic system definition.
2. successive refinement through analysis.
3. atomic implementation of work product sections.
4. successive implementation of higher level wps(s) through synthesis.
5. a hybrid (analysis and synthesis combined) step used to create two or more lower abstraction level wps(s) to implement two or more higher abstraction level wps(s).
6. finalization test.

Steps 2, 3, 4, and 5 do not necessarily need to be performed in sequence but can be interleaved. Step 5 can be used as a substitute for steps 3 and 4 when more than one implementor wps and more than one implementee wps are used. Obviously, there are similarities between our proposed framework and previous methods for system development, but there are also several differences. We elaborate on those similarities and differences at the end of this section after describing each of the steps in our method.

1. Basic System Definition: In this step we create two system wps(s) to match two meta-artifacts, which are the top ($\top$) and bottom ($\bot$) of our formal implementation model. These two wps(s) will represent the main requirement of the system and the system implementation itself. For example, in Figure 3 wps(s) *DBMSMainReq* and *DBMSImpl* correspond to $\top$ and $\bot$ respectively. Figure 2, Subfigure (1) depicts this step. Our final goals are: i) that the implemented system (represented by $\bot$) completely implements every functional and nonfunctional requirement and ii) that the system goal (represented by $\top$) is uniquely implemented by every deployed system component. The definition of a $\top$ requirement as a first step in developing a system is similar to Leveson's *Intent Specifications* method where the goals of the system are stated as the very first step [16, 17].

2. Analysis: In an analysis step we divide a wps into two or more wps(s) that will reside in a lower level of abstraction than the original wps. By construction each of the newly created wps(s) will partially implement the original higher abstraction level wps. For example, we can divide a requirement into two or more newly derived requirements or a system into two or more subsystems. Figure 2, Subfigure (2) illustrates this step.

An analysis step can be seen, in part, as stepwise refinement, in which case the verification of the higher level wps (implementee artifact) is the question weather the lower abstraction level wps(s) (implementor artifacts) completely provide all the behavior required or specified by its corresponding higher abstraction level wps. However, we need more than just complete implementation, we also need to assure that the lower level wps(s) do not implement any other unspecified behavior. Therefore, the additional restriction we apply to an analysis step is that the newly created wps(s) are implementation independent and completely specified. In other words, these newly created components collaborate in order to implement the higher level wps, but they do not implement each other nor collaborate to implement a behavior that is not explicitly stated by the higher level wps. Notice that we are asking that the newly created wps(s) be completely specified, not necessarily completely formally specified.

3. Atomic Implementation: In this step the atomic implementation of a work product section (wps) is performed. This is the basic implementation step; in it, an analyst or system engineer indicates that a given wps totally implements a given requirement or higher abstraction level wps. Figure 2, Subfigure (3) illustrates this step.

By *atomic* implementation we mean that given an implementee wps that has been sufficiently analyzed and specified in such a way that its implementor wps is a simple and direct implementation of its corresponding higher abstraction level implementee and no further analysis and separation is necessary. In other words, a rigorous argument of correct and total implementation can be made by a group of trained engineers without further analysis and separation.

4. Synthesis: In a synthesis step we unify two or more higher abstraction level wps(s) (atomic or synthesized) into one lower abstraction level wps. By construction the new wps partially implements those higher abstraction level wps(s) synthesized into it.

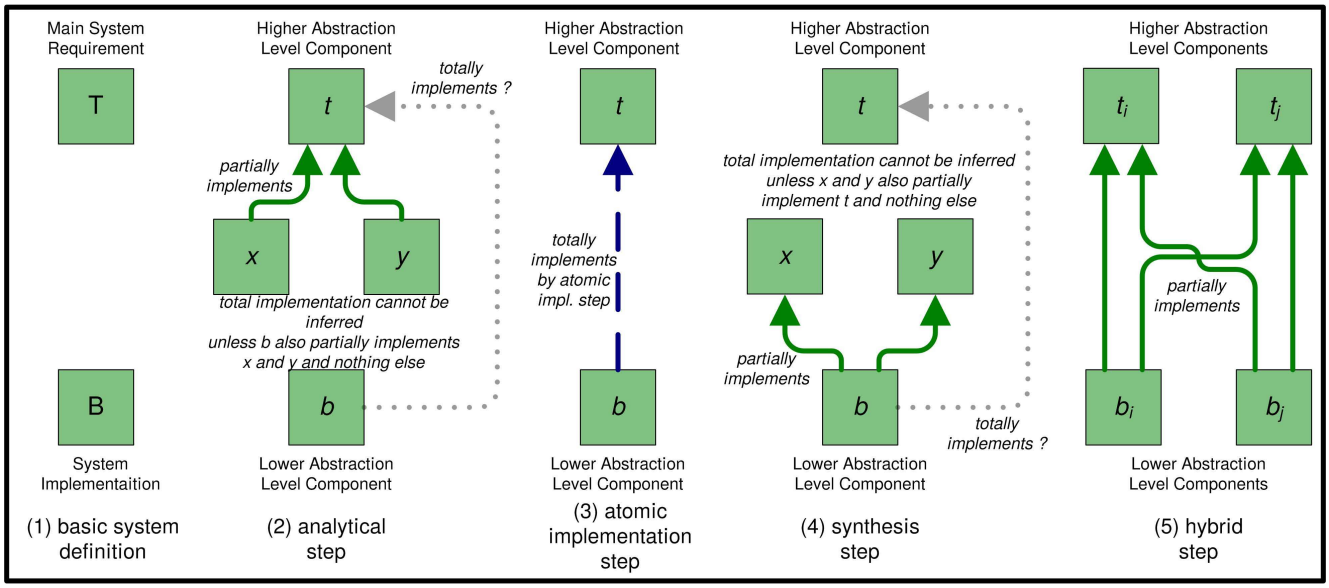

Figure 2. System Development through Formalized Analysis and Synthesis Implementation Steps.

The restriction on this synthesis step is that the new lower abstraction level wps does not add any new functionality. In other words, the new component partially implements only those wps(s) used in its composition. Figure 2, Subfigure (4) illustrates this step.

5. Hybrid Step (Analysis/Synthesis): Since analysis and synthesis steps can be performed together we add an additional step called a hybrid step, which corresponds to the combination of analysis and synthesis steps together. Figure 2, Subfigure (5) illustrates this step.

The rationale behind this step is twofold. First, it enables the modeling of wps reuse, that is, where one lower level wps is used to partially implement more than one higher level wps. Notice that this includes the reuse of any artifact not only source code. Second, the hybrid step enforces the following rule, which is a derivation of the conceptual completeness principle [5]: if wps b_i partially implement wps(s) t_i and t_j, and there exists wps b_j that partially implements t_i, then b_j must partially implement t_j as well.

6. Finalization Test: The development of the system is finalized when the system has been completely implemented and we can prove our goals by analyzing the formal implementation model of the system. That is the $\top$ wps is being totally implemented by every wps in the model and that the $\bot$ wps totally implements every wps in the formal implementation model. This with the exception of those wps(s) that belong to internal lattice-like sub structures created by the use of a hybrid step.

This finalization step is verified by testing that given any two wps(s) x and y in the formal implementation model (Γ) the set of lower bounds of (x, y) and the set of upper bounds in Γ are nonempty. As a result, a complete formal implementation model would contain a nonempty set of chains from $\bot$ to $\top$ where every wps belongs to at least one chain. The model is neither complete not valid until this condition is satisfied.

5.1. Comparison with Current Methods

Clearly, there are similarities between our proposed implementation-oriented secure architectures framework and current methods for system development.

Similarity (a): Similar Development Process. This framework strongly resembles the current state of the practice development process of requirements analysis, architectural design, implementation, and integration [11]. This similarity is in fact one of the advantages of our framework with respect to purely formal specification approaches.

Similarity (b): Similar Component Implementation Techniques. This framework does not specify which techniques, specification or programming languages need to be used for developing system artifacts. It only constrains the way resulting wps(s) are combined with each other in order to implement a higher level artifact.

Nevertheless, there are also differences.

Difference (a): Constrained Analysis and Synthesis Steps. In contrast with current state of the practice development methods, in this framework, analysis and synthesis steps are constrained by architectural constraint rules. For example, if an analysis step is performed and a higher abstraction level wps is divided into two lower abstraction

level wps(s), then there must be no remaining behavior in the higher abstraction level wps that has not been moved into the lower abstraction level. For example, if a module t that partially implements a requirement r is divided into two modules b_i and b_j, then module t must not partially implement any of the behavior specified by r; only b_i and b_j must, and the responsibilities of t will be reduced to the control of b_i and b_j.

Difference (b): Traceability Links Are Kept at all Stages of the Development Process. In this framework, and in contrast with both current state of the practice development methods and purely formal specification methods, we can and do trace which artifacts are implementing each other and how components or artifacts collaborate in order to implement a higher abstraction level artifact such as a requirement. This formal and complete traceability approach facilitates the verification and validation of system correctness with respect to the system's informal requirements and environmental constraints.

Difference (c): Formal Finalization Step. In this framework there is a clear and formal finalization step which is oriented toward the objective of proving that the final implemented system completely and uniquely implements the stated system goal. This step is a formal test in the architecture of the system and necessarily needs to be proved for a given formal implementation model of a system.

6. Vulnerability Case Studies

In order to describe how our implementation-oriented approach to secure architectures could help to build more secure systems, and to enable discussion about this framework, we introduce two vulnerabilities along with their associated formal implementation models. We also show how the application of our set of rules would have helped to anticipate these vulnerabilities.

6.1. Vulnerability Note VU#180147: Oracle DBMS Authentication Bypass

The Vulnerability Note VU#180147 was published by the US-CERT in February 2006 [23]. This vulnerability affects multiple versions (9i, 8i, 8) and multiple releases (from 8.0.1 to 9.0.1) of the Oracle® Database Management Server[2] (DBMS).

The Oracle DBMS, specifically the Database Server Application (*DBServerApp*), allows authorized users to remotely connect to the database and request the *DBServerApp* to execute a PL/SQL[3] statement. The process in charge of receiving PL/SQL queries or statements is called the *Listener* (Figure 3). Whenever the *Listener* receives a PL/SQL query or statement, it forks a new process (*ChildSQLProcess*), which is in charge of verifying that the requesting user has appropriate access rights in order to execute such a query/statement and also is in charge of executing the query/statement. However, if the user request contains a call to an external library, the *ChildSQLProcess* requests the *Listener* to execute the requested external library. In order to execute the external library the *Listener* calls an external library executor process *ExtLibraryExec*. The vulnerability resides in the fact that neither the *Listener* nor the *ExtLibraryExec* processes verify that the user is authorized to perform such a query/statement; only the child processes *ChildSQLProcess(es)* do. Therefore, a malicious process masquerading as a *ChildSQLProcess* can request the *Listener* to call an external library and completely bypass the authentication procedure [21, 23].

Figure 3 shows a formal implementation model we have developed to represent the requirements and components of a DBMS application. In Figure 3 the *DBMSMainReq* and *DBMSImpl* components are the $\top$ and $\bot$ meta-artifacts of our formal model, respectively. Also, the partial implementation traceability links from the *AuthLibs* component to every *ChildSQLProc[1/2/3]* component, from every *ChildSQLProc[1/2/3]* component to the *Listener*, and from the *Listener* to the *DBServerApp* can be derived from the architectural structure of the source code (i.e., a call tree). The traceability link from the *AuthLibs* component to the *AuthPolicy* requirement can be argued true by construction of the system; in other words, they can be added by an atomic implementation step. The traceability links from *Listener* to *AuthPolicy* as well as the traceability link from *DBServerApp* to *AuthPolicy* cannot be argued by construction since non elementary, analysis, implementation, and synthesis steps have been used for the development of the lower level components; they need to be proved valid in the formal implementation model.

Observe the dotted arrow from the *AuthLibs* component, which represents the implementation of the authorization policy, into the *ExtLibraryExec* component. This dotted arrow indicates that the *ExtLibraryExec* component does not partially implement *AuthLibs*. In this implementation-oriented secure architectures framework the absence of this traceability link represents a security vulnerability. This is because the *Listener* component does not completely implement the requirement *AuthPolicy* (due to the missing traceability link represented by the dotted arrow). As a result we cannot infer that *DBServerApp* totally implements

[2] Oracle® is a registered trademark of the Oracle Corporation.

[3] PL/SQL (Procedural Language/Structured Query Language) is a proprietary extension to the SQL language used in Oracle® Database Management Systems (DBMSs). PL/SQL allows procedural and object-oriented constructs to be used along with embedded SQL statements or queries for the access and management of data in an Oracle® DBMS [20].

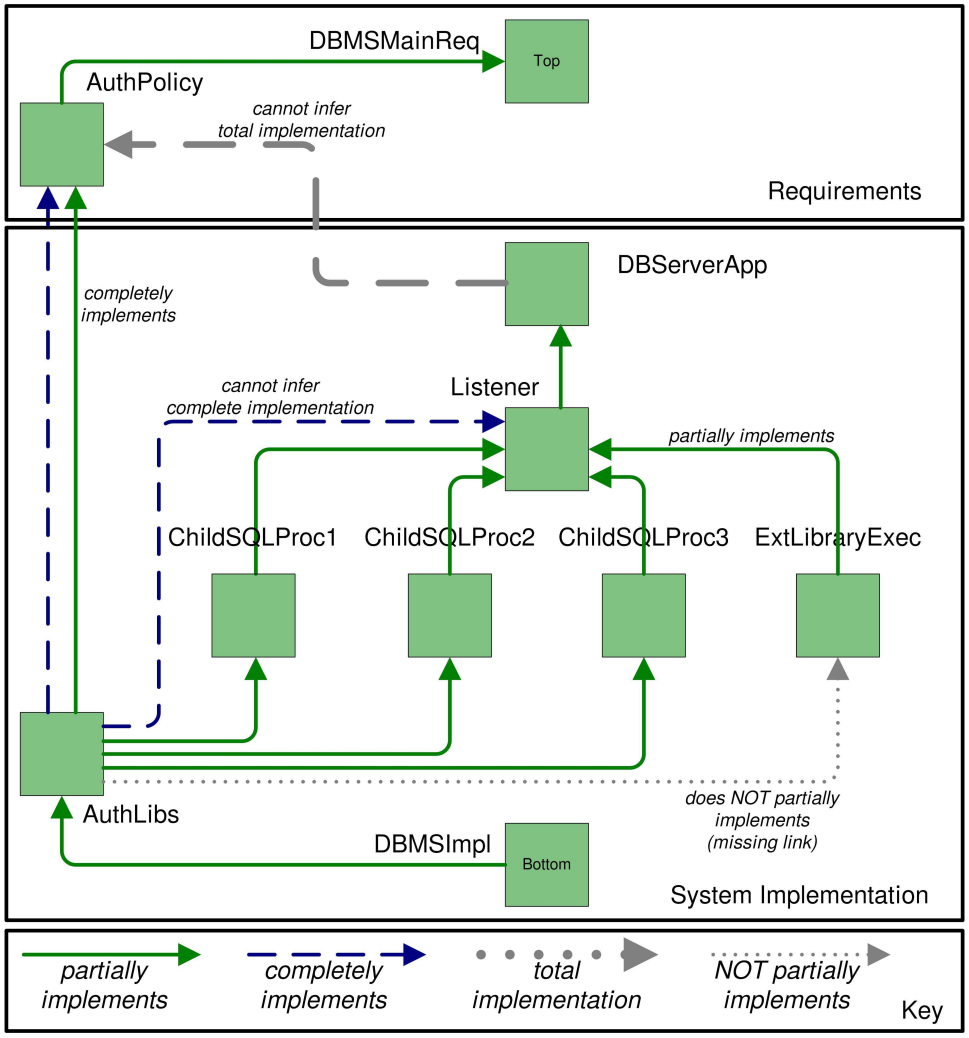

Figure 3. Partial Formal Implementation Model and Associated Security Vulnerability (Example 1).

the corresponding *DBMSMainReq*, which is our final goal.

In previous work [5], which focused on the discovery of hidden implementation dependencies in order to anticipate safety risks, the missing traceability link from *AuthLibs* to *ExtLibraryExec*, among others, is called a hidden implementation dependency and was discovered and indicated as an *impDependecy* traceability link by our SyModEx tool.

6.2. Vulnerability Note VU#871756: Oracle DBMS Escalation of Privileges during Initial Authentication

The Vulnerability Note VU#871756 was published by the US-CERT on 17 January 2006 [24]. This vulnerability affects multiple versions (10g, 9i, and 8i) of the Oracle DBMS [10].

The user authentication protocol to connect to an Oracle DBMS is composed of two steps. In the first step the client process sends a username to the server process, and the server validates the username and responds to the client. In the second step the client sends the username, an obfuscated password, and a set of attribute/value pairs to configure the user session. One of these attribute/value pairs is the "*AUTH-ALTER-SESSION*" attribute, which is associated with an SQL statement. The security vulnerability VU#871756 resided in the fact that this SQL statement was executed by the *DBMSServerApp* without any verification whatsoever of user rights, allowing for any valid user to execute an arbitrary SQL statement with system privileges.

Stated in the terms of our architectural implementation model (Figure 4), the process in charge of executing the SQL statement (*SetUpSessionAttributes*), or one or more of its lower abstraction level implementor wps(s), is (are) not partially implemented by the authentication libraries *Auth-*

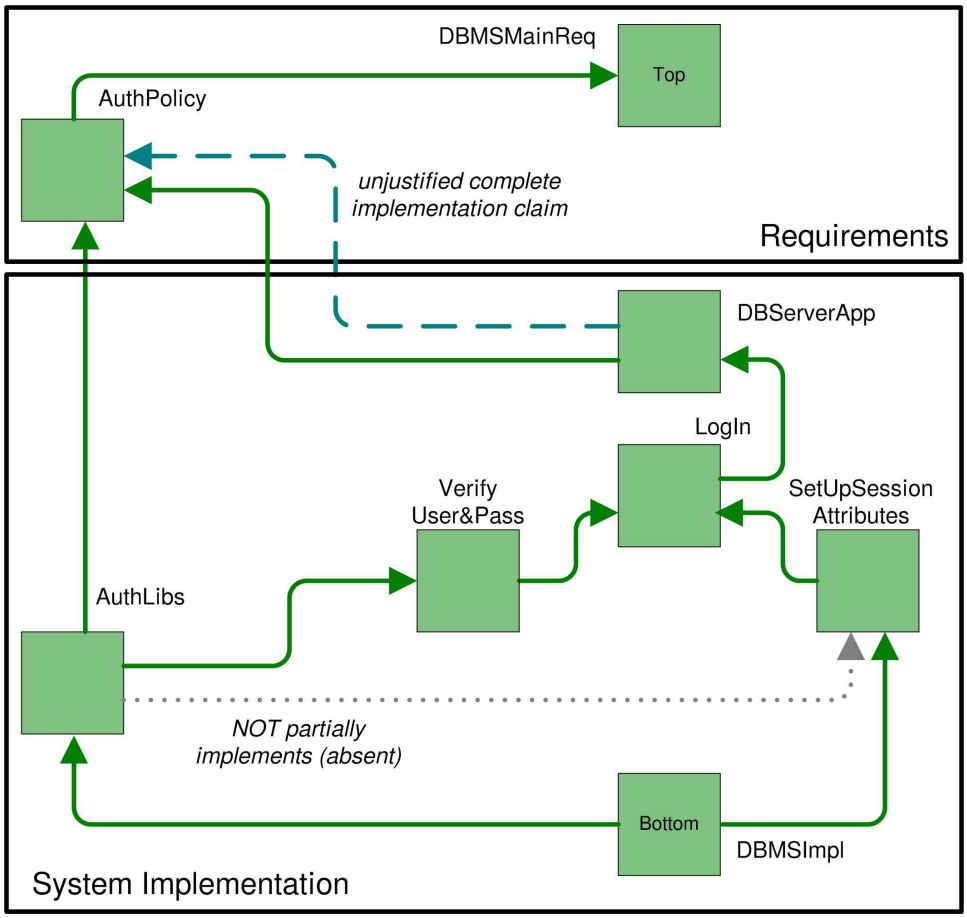

Figure 4. Partial Formal Implementation Model and Associated Security Vulnerability (Example 2).

Libs. Therefore, the *LogIn* process does not totally implement the authentication policy *AuthPolicy*. Figure 4 shows a formal implementation model developed to represent this vulnerability.

We remark that the models shown in Figures 3 and 4 are only simplified interpretations of an architecture corresponding to the Oracle® DBMS at the time of the vulnerabilities. We do not imply in any way that these models represent the actual architecture of the Oracle® DBMS. The models were developed only for the purpose of exemplifying how our framework could help minimize the occurrence of security vulnerabilities.

7. Conclusions

We introduced a framework composed of i) a formal implementation model of a system, which is based on set theory and predicate calculus [4, 5] and ii) a method for the successive refinement of a formal implementation model of a system through analysis and synthesis steps. These formalized development steps constrain the formal implementation model architecture of a system to a lattice-like algebraic structure. This algebraic structure ensures that all requirements and constraints that a system component must implement are in fact considered when developing the system. Furthermore, this framework enables rigorous verification and validation of system correctness while at the same time allows for informal approaches to be used in a more systematic way.

We believe that our framework is a step toward the effective use of mathematically-based approaches in state of the practice software engineering. Its main strength resides in the fact that it enforces an algebraic structure to a system architecture, while at the same time, allows the formalization level to be adjusted during system development and to vary depending on the portion of the system in question. However, we also recognize that the framework is not without problems.

In order to further enable a proactive discussion of our implementation-oriented approach to secure system architectures, we now introduce some topics and questions which we believe need to be considered and addressed as a

prerequisite for a successful implementation of our framework.

One difficulty we anticipate for the future implementation of our approach is the transformation step from old development methods to the framework introduced in this paper. How do the object-oriented design and programming methods interact with our new proposed framework? Is it possible to implement a system created using our implementation-oriented secure architectures using object-oriented design and programming techniques? What changes may our framework need to undergo in order to coexist with the object-oriented methodology?

In a similar way, what changes will current programming languages (structured and object-oriented) need to undergo in order to accommodate the framework? For example, when implementing an artifact we require that all functionality must reside in the lower abstraction level (implementor) wps and that the higher abstraction level artifact (implementee) must act as manager of the lower level wps(s). What changes would be needed to programming languages in order to accommodate these restrictions? As one of many options we could, for example, require code modules to be segregated into either control modules or functionality modules.

This framework has been purposely designed with the characteristic of allowing the modeling of a main system goal and the system requirements (including functional and nonfunctional requirements) from the start. Furthermore, it has been designed to allow a formal representation of the partial implementation traceability links between the system requirements and all other artifacts. For this reason, we believe in the adequacy of the framework to model all security features of a system. This is because, in the end, all security features must necessarily derive from, or map to, a security requirement or constraint.

For example, we believe that most current buffer overflow vulnerabilities could be seen, at a high level of abstraction, as the absence of a partial implementation link between a wps representing a compiler and a protection constraint wps stating that variables must not be referenced outside their memory boundaries. Of course, the same constraint could be totally implemented by other wps(s) more internal to the lattice-like formal implementation model; for example, by using a safe library that ensures (completely implements) such constraint or even by ensuring that every access to a variable partially implements such constraint.

Although we believe that by using our framework most security features could be modeled and verified at a workable level of abstraction, we also recognize that more work is needed in order to address the scalability of the process.

8. Related Work

The idea of stating system goals up-front has been borrowed from the work of Leveson and colleagues at the Massachusetts Institute of Technology. Leveson et al. have developed a safety and human-centered approach to the design and specification of safe systems (air traffic control and airborne systems) [15, 16]. The basic techniques behind Leveson's approach are twofold: *Intent Specifications* and the *SpecTRM-RL* requirements specification language. The first step in an *Intent Specifications*-based specification is the statement of system goals (intent). Our framework goes a step further in this approach and states that there is one system goal that encompasses all other system objectives or subgoals. We call this goal the *Main System Requirement* and it forms the $\top$ (top) meta-artifact of our architecture. In a similar way the highest refinement level in an *Intent Specification* is the system implementation level, which we unified into our *System Implementation* component corresponding to our $\perp$ (bottom) meta-artifact.

With respect to formal approaches that allow the development of software implementations while keeping full traceability to the formal structures, the Krestel Institute offers a full-blown development application based on the use of their *Metaslang* formal specification language and the use of categorical operators for the derivation of program refinements. Krestel's formalization approach defines what is called the Specware development process and commences with an initial formal specification written in *Metaslang*. Successive refinements are applied to the specification until it reaches a final state; all stages of the specification are executable [12]. However, this approach necessarily requires the use of the *Metaslang* formal specification language and of proprietary component implementation technologies. In contrast, our framework allows the use of any specification and programming language as long as the system architecture is constrained by its formal implementation model as described in this paper.

9. Acknowledgments

The material presented in this paper is based on research sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number F30602-02-1-0178 with the Center for Secure and Dependable Systems (CSDS) at the University of Idaho. 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 and DARPA or the U.S. Government.

10. References

[1] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable and Secure Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[2] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, ser. SEI Series in Soft. Eng. New York, NY, U.S.A.: Addison-Wesley, 2002.

[3] D. Conte de Leon, "Formalizing traceability among software work products," Thesis, Univ. of Idaho, Moscow, ID, U.S.A., Dec. 2002.

[4] D. Conte de Leon, "Completeness of implementation traceability for the development of high assurance and critical computing systems," Dissertation, Univ. of Idaho, Moscow, ID, U.S.A., 2006.

[5] D. Conte de Leon and J. Alves-Foss, "Hidden implementation dependencies in high assurance and critical computing systems," *IEEE Trans. Softw. Eng.*, In press, 2006.

[6] R. I. Damper, "Emergence and levels of abstraction," *Int'l Journal of Systems Science*, vol. 31, no. 7, pp. 811–818, Jul. 2000.

[7] "Galois Haskell-based approach to high assurance systems," Galois Inc., Beaverton, OR, U.S.A., 2006. [Online]. Available: http://www.galois.com/

[8] A. Hall and R. Chapman, "Correctness by construction: Developing a commercial secure system," *IEEE Software*, vol. 19, no. 1, pp. 18–25, Jan./Feb. 2002.

[9] H. M. Hinton, "Under-specification, composition and emergent properties," in *Proc. of the 1997 Workshop on New Security Paradigms (NSPW '97)*. New York, NY, U.S.A.: ACM Press, 1997, pp. 83–93.

[10] "Oracle® DBMS - critical access control bypass in login bug," IMPERVA®, Foster City, CA, U.S.A., Jan. 2006. [Online]. Available: http://www.securityfocus.com/bid/4033

[11] Institute of Electrical and Electronics Engineers, "Guide to the software engineering body of knowledge," IEEE, Tech. Rep. Ironman Version, Jun. 2004.

[12] Kestrel, "Specware 4.1 tutorial," Kestrel Development Corporation and Kestrel Technology LLC, Tech. Rep., 2004.

[13] "Specware®," Kestrel Institute, Palo Alto, CA, U.S.A., 2004. [Online]. Available: http://www.specware.org/

[14] J. Launchbury, "Galois: High assurance software," in *Proc. of the 9th. ACM SIGPLAN International Conference on Functional Programming*. New York, NY, U.S.A.: ACM Press, 2004, p. 3.

[15] N. G. Leveson, M. de Villepin, M. Daouk, J. Bellingham, J. Srinivasan, N. Neogi, E. Bachelder, N. Pilon, and G. Flynn, "A safety and human-centered approach to developing new air traffic management tools," Aeronautics and Astronautics Department, MIT, and Eurocontrol Experimental Centre, Tech. Rep., Dec. 2001.

[16] N. G. Leveson, "Intent specifications: An approach to building human-centered specifications," *IEEE Trans. Softw. Eng.*, vol. 26, no. 01, pp. 15–35, Jan. 2000.

[17] N. G. Leveson, "A systems-theoretic approach to safety in software intensive systems," *IEEE Trans. Dependable and Secure Comput.*, vol. 1, no. 1, pp. 66–86, Jan. 2004.

[18] N. G. Leveson, *System Safety Engineering: Back to the Future*. Draft of Book, Jun. 2006. [Online]. Available: http://sunnyday.mit.edu/book2.pdf

[19] "Praxis High Integrity Systems," Praxis High Integrity Systems Limited, Bath, U.K., 2006. [Online]. Available: http://www.praxis-his.com/

[20] J. Price, *Oracle Database 10g SQL*. Hightstown, NJ, U.S.A.: McGraw Hill, 2004.

[21] "Oracle TNS listener arbitrary library call execution vulnerability," Security Focus™, Calgary, AB, Canada, Feb. 2002. [Online]. Available: http://www.securityfocus.com/bid/4033

[22] Software Engineering Institute, "Capability maturity model integration for systems engineering, software engineering, integrated product and process development, and supplier sourcing," Carnegie Mellon Univ., Pittsburgh, PA, U.S.A., Tech. Rep. CMU/SEI-2002-TR-012, Mar. 2002.

[23] "Vulnerability note VU#180147," United States Computer Emergency Readiness Team (US-CERT®), Arlington, VA, U.S.A., Feb. 2002. [Online]. Available: http://www.kb.cert.org/vuls/id/180147

[24] "Vulnerability note VU#871756," United States Computer Emergency Readiness Team (US-CERT®), Arlington, VA, U.S.A., Jan. 2006. [Online]. Available: http://www.kb.cert.org/vuls/id/871756

[25] E. N. Zalta, Ed., *The Stanford Encyclopedia of Philosophy*. Stanford, CA, U.S.A.: The Metaphysics Research Lab Center for the Study of Language and Information, Mar. 2006, ch. Emergent Properties. [Online]. Available: http://plato.stanford.edu/entries/properties-emergent/