Software Engineering for Security: Towards Architecting Secure Software

Doshi Shreyas
Information and Computer Science Dept.
University of California, Irvine
CA 92697, USA.
+1 949 824 8438
doshi@acm.org

ABSTRACT

Since the advent of distributed systems, security of software systems has been an issue of immense concern. Traditionally, security is incorporated in a software system after all the functional requirements have been addressed. This paper argues for the need for security concerns to be an integral part of the entire software development life cycle. Different research areas that lie at the confluence of Software Engineering and Security are surveyed. Finally, the paper focuses on the use of Software Architecture to solve certain problems that are faced in the engineering of secure systems.

Keywords

Software Engineering, Security, Software Architecture, Secure Software Architecture.

1. INTRODUCTION

As society becomes more and more reliant on software systems for its smooth functioning, software security is emerging as an important concern to many in the field of Computer Science. Since security attacks can cause anything from losses worth millions of dollars in business to intrusion into defense systems, the repercussions of such security attacks can be quite grave. However, it is not only large organizations and governments that are susceptible to security attacks. Today, security has also become a concern for the average citizen. Citizens are becoming increasingly aware about the security threats over computer networks, encouraging them to take adequate steps to protect their credit card numbers and personal information over the Internet. Though such preventive steps by ordinary citizens and organizations are necessary, they do not offer long-term solutions to the problem of security of software systems. It is known fact that the *wily hacker* [CB94] can find ways to get around these steps. What then is the solution?

Doshi Shreyas
Term paper for ICS 121

Much of the work so far in the area of security of software systems has come from the Cryptography community. While other areas of Computer Science like Computer Networks and Theory have also contributed to the solution to the problem of security of software systems, we found that the area of Software Engineering has made very little contribution. This is quite ironic considering the fact that the problem is that of how 'software' systems can be made more secure. These software systems are an outgrowth of some Software Engineering process presumably with Software Engineering principles applied in their development. It is fair to deduce that we might be able to make systems more secure by incorporating security considerations explicitly in the Software Engineering process and by applying certain Software Engineering principles to solve problems faced in the engineering of secure software. However, several impediments lie along this path.

Security is a non-functional requirement [CN95]. Since developers grapple with the problem of getting the functionality right without overrunning schedules or budgets, security is not the utmost concern for system developers- even in systems where security threats might be easily perceptible. Hence, software system security is typically an afterthought [Gas88], i.e. security may be considered seriously if the functional requirements are met and the project is within the schedule and budgets. As observed in [Bro87], this is seldom, if ever, the case.

In this paper, we shall look at the different research areas that lie at the confluence of the fields of Software Engineering and Security. More specifically, we shall look in detail how research in the area of Software Architecture [PW92] can help solve the problems that lie in the path of development of secure software systems. Before we delve further into the topic, it is worthwhile to examine what security really means and the different dimensions of security.

Security of a software system is a multi-dimensional concept [BSW01]. The multiple dimensions of security are:

- *Authentication*- The process of verifying the identity claimed by an entity.
- *Access control*- The process of regulation of the kind of access (e.g.- read access, write access, no access) an entity has to the system resources.
- *Audit trail*- A chronological record of events leading up to a specific security-relevant system state. This record can later help in the examination or reconstruction of the specific security scenario of interest.
- *Confidentiality*- The property that deals with making certain information unavailable to certain entities.
- *Integrity*- The property that information has not been modified since its inception from the source.
- *Availability*- The property of the system being accessible and usable for an authorized entity.
- *Non-repudiation*- The property that places confidence regarding an entity's involvement in certain communication.

Security can mean different things at different times. Generally, when security is referred to, it essentially implies one or more of the above dimensions of security. For example

- Security in E-mail communication might involve ensuring confidentiality, non-repudiation and integrity.
- Security in online shopping would involve ensuring authentication, confidentiality, integrity and non-repudiation.

A security *attack* (or simply an attack) is an attempt to adversely affect one or more of the above security dimensions of the system. When an attack is successful, the security of a system is said to have been compromised. Throughout this paper we use the term *secure system*. We define a secure system as one that has the requisite type and amount of security to be able to counter the potential threats it may face. In the remainder of the paper, we shall use the above security-related terms frequently. We shall also use certain terms from the related area of cryptography. Readers unfamiliar with the basic terms in cryptography are referred to [Sch96].

This paper is organized as follows. In section 2, we identify and classify research areas in the fields of Software Engineering and Security. In section 3, we look at the use of software architecture for engineering secure software systems. Section 4 presents a summary of the contributions of this paper and section 5 concludes the paper by raising some points for discussion.

2. SOFTWARE ENGINEERING AND SECURITY

We categorize research in the area of Software Engineering and Security into two research directions-

- *Software Engineering for Security*- This area explores Software Engineering research and principles that can be used in the engineering of secure systems, or to enhance the security of software systems.
- *Security for Software Engineering*- This refers to research in the areas of security and cryptography that helps solve problems in different areas of Software Engineering.

We shall now look at each of the above categories and also identify research directions that lie therein.

2.1 Software Engineering for Security

Software Engineering research for Security can roughly be structured around the waterfall model as follows.

- *Security requirements engineering*- Eliciting security requirements is a critical step in adopting a security-oriented software development approach early in the life cycle. Security requirements should be determined after the functional requirements have been ascertained [DS00], since they are intimately dependent on the kind of functional requirements that have been gathered. Generic security requirements for systems have taken the form of security models or policies [BP75] [GM82] [McL94]. More recent work [VKP01] has attempted to study *trust* assumptions of software developers regarding how the users will use the system. According to [VKP01], erroneous trust assumptions of developers regarding the manner in which the system will be used usually result in compromise of security. Though developers make such assumptions throughout the software development life cycle, the two main reasons for dangerous and erroneous trust assumptions are:
  - Incomplete requirements and
  - Miscommunication between developers.

  Hence, it is necessary to elicit accurate trust assumptions during the requirements analysis phase. We believe that research in the area of requirements engineering for security needs to be focused in this direction. Such an approach could also be useful to the developer of a secure system in that the identification and articulation of trust assumptions can be used as a security guideline for later stages of software development.

- *Formal Analysis of security protocols*- The security protocols that developers choose to use in their system are not necessarily as secure as the developers might perceive them to be. For example, flaws have been found in almost every authentication protocol that has been published to date. Hence, we believe that before developers can place trust in a particular security protocol, it is necessary that the trust be justified explicitly and formally. This can be done only by a formal analysis of candidate security protocols. For example, [Kem89] describes how an encryption protocol can be formally analyzed using the Ina Jo [LSS+80] specification language. While Ina Jo is a generic specification language, several

techniques have been developed specifically for formal analysis of security protocols (e.g.- [BAN90]). We believe that it is necessary for software engineers to make a decision about the security protocol they will use in their system only after formal analysis of the protocol puts sufficient confidence in the correctness and security of the protocol. Such formal analysis could be done by a trusted external agency or by the software engineers themselves.

- *Architecting/Designing secure software*- According to [DS00] there is an emergent need for integrating security policies with the design of systems. For example, a research direction would be [DS00] to extend standards such as UML [RSC97] so as to include the explicit modeling of security dimensions like authentication, access control, etc. identified earlier. We believe that software architectures can play a vital role in the development of secure systems. The use of principles of Software Architecture in solving security-related problems in distributed systems and in making software systems more secure forms the focus of this paper.

- *Programming languages and programming paradigms for secure software*- According to [VBC01], programming language designers have until recently ignored security primitives that programmers should have at their disposal. In fact, security risks are known to have been uncovered in several programming languages (e.g.- C, C++ [VBC01]). In this regard, the Java security architectural extensions [WBD+97] [Gon97] are a step in the right direction. A related development in this area is the emergence of the aspect-oriented programming paradigm (AOP) [KIL+97]. The object oriented programming paradigm models security very poorly since invocations to security methods are typically scattered throughout the application code, making it difficult to abstract the security aspect of software. AOP can provide a separation of the security aspect of the system from all other aspects like reliability, fault-tolerance, etc. This can be very useful in programming security into an application, since all security concerns can be programmed together and later dispersed through the actual code using an *aspect weaver* [KIL+97]. Emerging technologies are posing several challenges to programming languages, requiring them to adapt their security mechanisms so as to secure the applications running on the technologies. For example, [GS01] describes the security risks in WML posed by the mobile e-commerce technology.

- *Security testing*- For any system where security concerns are sufficiently large, it is necessary to test the system to determine explicitly whether it satisfies the security requirements. We believe that testing for functionality is significantly different from testing for security. This is because security, unlike functionality, is not an externally observable property. Hence, we believe that existing testing strategies that work well for testing functionality need to be extended in order to accommodate explicit security testing. A conventional approach to security testing has been the hiring of a security expert who would try to attack the system and exploit any potential weaknesses of the system. Though this approach is a very attractive one, we believe that more needs to be done. Since testing is a phase during which budget and schedules are tight, it is necessary to devise automated security testing mechanisms which can test the security of the system with less effort (time and cost) than the manual security testing approach described above. For example, [FL94] and [FKA+94] describe a semi-automated approach for security testing. [DWW99] describes the reasons for not relying on conventional testing strategies for testing security and recommends the promotion of open security testing to increase confidence in the security of software.

We now look at how research in the area of security can help solve certain problems in software engineering.

2.2 Security for Software Engineering

Security can solve the following problems faced in Software Engineering.

- *Secure Software Deployment and Configuration Management*- Several privacy and security issues [DGS99] surface in post-deployment configuration management [HW96]. [DGS99] discusses these issues and attempts to find solutions to the problems faced in this area. According to [DGS99], cryptographic techniques emerge as an important solution to these issues.

- *Component test coverage claims*- Vendors of Commercial Off the Shelf (COTS) components are faced with the challenge of assuring their customers that adequate testing has been done on their components without actually revealing source code, test cases, etc. Revealing this might convey vital information about the component. If such information is revealed, it is possible that the component may be created by a competitor thereby making the vendor lose its competitive advantage. In [DS99], a cryptographic technique by which the test coverage claims of a component can be verified without revealing essential information about the component is described.

- *Protection of software*- Piracy of software is a source of tremendous losses to the software industry. Hence, efforts to effectively counter the software privacy problem are necessary. Both software (e.g.- [HP87]) and hardware-based (e.g.-[MM84]) solutions to the

problem have been proposed. Most of the software-based approaches proposed this far use cryptography. So far, however, not many solutions to the problem have been effective because of the adversarial economics [DS00] involved.

In this section, we looked at the confluence of the areas of Software Engineering and Security from two perspectives:

- Software Engineering for Security and
- Security for Software Engineering.

To engineer secure software, the first perspective is more useful and we shall be looking at this perspective in the remainder of this paper. Within this perspective, we shall be focusing on the use of Software Architecture for engineering secure software.

3. SOFTWARE ARCHITECTURE FOR SECURITY

We believe that research in the field of Software Architecture can help solve several problems that lie in the path of developing secure software. In the next subsection, we describe CORBA security as an example of the security considerations involved in a component-based distributed system. We describe in sections 3.2 through 3.6 several aspects of software architecture and how they can be used to tackle specific problems in engineering secure software. Finally, in subsection 3.7 we look at how the convergence of Aspect-Oriented Programming (AOP) and Software Architecture may be a promising area for future research.

3.1 CORBA Security

According to [WWW1], Common Object Request Broker Architecture (CORBA) is an architecture and specification for creating, distributing, and managing distributed program objects in a network. It allows programs that may be developed by different vendors and may be situated at different locations to communicate in a network through an *interface broker* [WWW2]. Various security issues emerge in a scenario described above. Here we will describe the basic Object Management Architecture and ORB architecture. With these foundations, we shall describe the security requirements in CORBA and the possible solutions to the security requirements. This will serve to articulate the security issues involved in Software Architecture and describe at a high level how they can be solved. [Chi98] gives further details regarding the CORBA Security Service. More information about CORBA may be found in [WWW2].

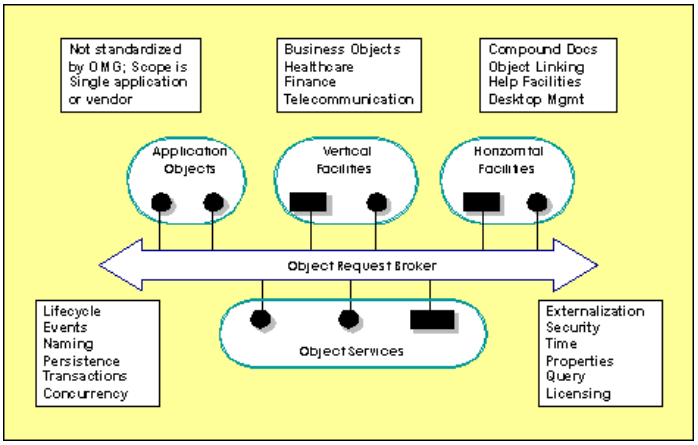

Figure 1. Object Management Architecture, printed from [Chi98]

Figure 1 shows a model of distributed object computing with the Object Management Architecture (OMA) of CORBA. At the center lies the Object Request Broker (ORB) which serves the purpose of connecting heterogeneous software components in arbitrary configurations. Four basic types of software components are identified in OMA:

- *Object Services*- provides basic services that are needed by other components in the distributed system.
- *Horizontal Facilities*- that are needed through by all users in the distributed system and may be used by components in the vertical facilities.
- *Vertical Facilities*- that provide capabilities for specific types of businesses.
- *Application objects*- which combine all other components and provide enterprise-specific services.

The need for developing a specification for the Object Request Broker resulted in the creation of the Common Object Request Broker Architecture or CORBA. Since clients and objects in a distributed system may be heterogeneous, it is necessary to devise means by which they may be able interact in a language and platform independent manner. In order to accomplish this, the interface to objects is defined using a standard OMG Interface Definition Language (IDL). Using such a standard interface, it is conceivable that clients and object implementations would be able to translate transparently between different programming languages, operating systems, data formats, etc. Such a scheme is depicted in figure 2.

Table a. Security issues and possible countermeasures in CORBA

Issue	Countermeasures
Authorized user gaining access to unauthorized information	Control of access to ▪ Interfaces ▪ Subsets of implementation of an interface ▪ Interface operations Non-repudiation measures Security audit log mechanism
A user masquerading as another user	User authentication and mutual authentication between the client and the object implementation.
Eavesdropping on a communication channel and tampering with communication between objects	Cryptographic measures like encryption and hashing
Bypass of security controls	Delegation mechanisms controlled by clients and object implementations

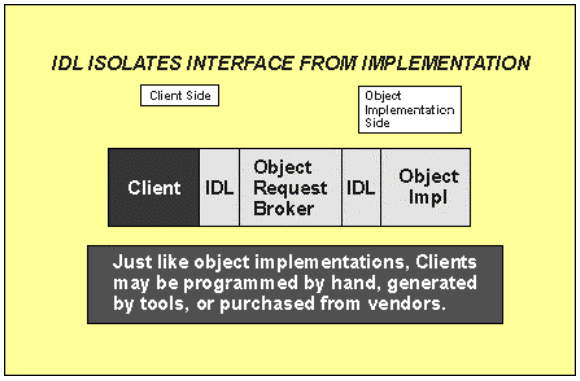

Figure 2. OMG IDL in CORBA, printed from [Chi98]

As described in [Chi98], the major security issues that emerge in case of the architecture described above are shown in Table a.

3.2 The Role of Software Architecture

Having seen the possible countermeasures against the security problems that arise in component based distributed systems, we will describe here specific ways in which we can deal with them by exploiting existing features of Software Architecture and by proposing new features, when necessary. The field of software architecture [PW92] provides a clear separation between components and their interactions. In this framework, an architecture description language (ADL) allows the specification of system in terms of the following abstractions [BI97]:

1. *Components*- define computational units written in any programming language.
2. *Connectors*- describe the type of interactions between components.
3. *Configuration*- defines a system structure in terms of interconnection of components through connectors.

Software architecture can be extremely helpful in engineering secure software systems in three distinct ways, depending on the scenario of the problem at hand.

- Definition of secure architectures using an ADL [MQR+97] [ML97].
- Integrating security for COTS based systems [BS98] [BS99].
- Resolution of complex security-related interactions between heterogeneous software components [BI97].

Apart from these benefits, use of principles of software architecture also provides the following additional benefits to the developer of a secure system

- Reuse of security-related code across different systems [JH98] [Dam98]
- Use of architectural patterns [YB97]
- Creation of autonomous security agents [FL96] [BGS+98] [QS98] [TOH99]

In the following subsections, we shall describe representative research from each of the above areas. We also look at a related topic in this section: the use of aspect oriented programming [KIL+97] in abstracting the security *aspect* of systems.

3.3 Definition of secure architectures

A promising approach to secure system design is the incorporation of security considerations into the software architecture. Software architecture can be used to define and model the security requirements of a software system in order to assist in

- Subsequent system development and
- Checking the implementation for compliance with the security requirements.

In order to model security requirements in the form of architecture, we may express them using an ADL. The problem of using an ADL to include security considerations in the overall software architecture has been addressed in [MQR+97] and [ML97]. In [MQR+97], the authors Moriconi et al. describe the use of SADL (an Architecture Description Language) in the formalization

of their Secure Distributed Transaction Processing (SDTP) system. The authors use a formal approach to software architecture [AG94] to incorporate security requirements by means of three steps [MQR+97]:

1. Formalizing the system architecture in terms of common architectural abstractions.
2. Specialization of the system architecture into different architectures, each depending on different assumptions regarding the security of system components.
3. Proving that every implementation corresponds to the system architecture or one of its specializations and thereby satisfies the security requirements.

In this subsection, we will discuss in detail, the approach described in [ML97] by Meldal and Luckham. In [ML97] the authors describe the use of the RAPIDE ADL [LV95] to define a *reference architecture* for the description of NSA's MISSI (Multilevel Information System Security Initiative) architecture. According to [ML97]

> *A reference architecture is an architecture used to define references against which implementations can be checked for compliance.*

The use of reference architectures is useful in that it provides an elegant way to obtain separation of concerns in the system. That is, one reference architecture for a system may be used to specify the security requirements of the system, while another may be used to specify its fault-tolerance requirements. We shall see later in this section how similar separation of concerns can be obtained using the aspect-oriented programming paradigm [KIL+97].

According to [LV95]

> *RAPIDE is a concurrent event-based simulation language for defining and simulating the behavior of software architectures.*

The authors use RAPIDE to capture two aspects of the MISSI reference architecture:

1. *Structures*- describes the arrangement of the different components of the system at the following levels of abstraction
   - Global level focuses on the main components and constraints on the interactions between the components.
   - The concept of operations or *conops* level focuses on the functional decomposition of the architecture.
   - The execution level focuses on the dynamic, physical structure of the system
2. *Information flow integrity*- describes the adherence of the interactions to certain policies and procedures as determined by the reference architecture.

The formal capture of the architecture, according to [ML97] involves three steps.

1. Identifying components
2. Identifying how they are connected and
3. Identifying how the connections are used.

These steps are applied to the system structures successively at the three levels of abstraction defined above: the global, conops and execution levels. The first two steps are self-explanatory, however, the last step deserves explanation. By identifying how the connections are used, it is possible to determine operational security constraints on the components and the their interactions. This is a convenient way of specifying for example, the conditions under which an interaction will occur between components and the conditions under which an interaction will not occur. Figures 3, 4 and 5 depict the application of steps 1, 2 and 3 respectively on the MISSI reference architecture at the global level of abstraction. The reader is referred to [ML97] for the details and explanation.

```
architecture MISSI( ) is
    internet  : WAN;
    DNS       : DirectoryServiceAgent;
    enclaves  : set(MISSI_Enclave);
    sites     : set(Site);
    ...
end MISSI
```

Figure 3. Components of the MISSI Reference Architecture, printed from [ML97]

```
connect
    for e: Enclave in enclaves.enum() generate
        internet.socket to e.wan_conn;
end;
```

Figure 4. Connecting Architectural Components, printed from [ML97]

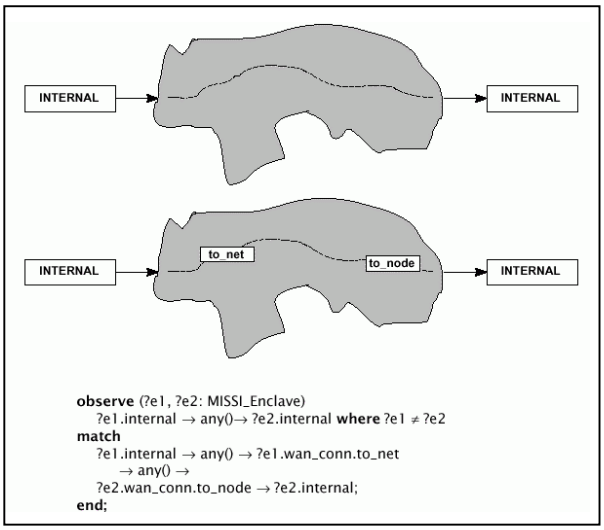

Figure 5. Security Constraint, printed from [ML97]

Once the three steps are applied at different levels of abstraction, *RAPIDE maps* can be used to relate components and interactions at one level of abstraction to components and interactions at another level of abstraction. The approach outlined here provides a

convenient way of specifying the security aspects of a system. The following benefits may be accrued by an approach of the kind described above:

- Convenient separation of the security-related aspects of the system
- Explicit formal modeling of the security-related aspects in the form of the software architecture
- Verification of the implementation with respect to the architecture and the verification of the architecture with respect to the requirements, by means of proofs.
- Providing a model to reason about the security properties of the system.

We shall now look at another problem; that of integrating security into COTS based systems.

3.4 Integrating security in COTS based systems

The use of Commercial off the shelf software (COTS) components is a very attractive choice faced by software development organizations for economical software development and reducing the time-to-market. Since COTS components are intended to work in varied environments that may have different security requirements, security is not typically not 'programmed into' these components. Hence, no or very limited generic security services are typically provided by COTS components. However, since most of these components are meant to work in a distributed environment, they system using these components would be susceptible to various security risks. It is therefore necessary to somehow incorporate security for the particular system at hand that uses these components.

This problem presents another area of research for the use of Software Architecture for engineering secure software systems. In fact, the problem is exacerbated due to the fact that the non-testability of some COTS components might actually pose major security threats to the system that uses such components [MV99]. [BS98] and [BS99] address the problem architectural approaches for integrating security in COTS based distributed systems. In [BS98] the authors describe the CERT HLA/RTI distributed interactive simulation environment that is developed using COTS components with security integrated into the system later. The RTI architecture is depicted in figure 6.

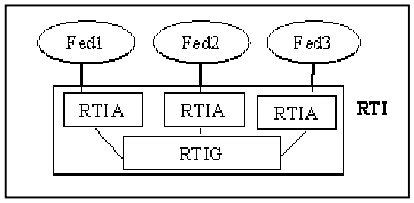

Figure 6. RTI architecture, printed from [BS99]

In the architecture, a federate (e.g.- Fed1) denotes an individual simulation. Each federate interacts with the RTI Ambassador (RTIA) component which exchanges messages over the network, in particular with the RTI Gateway (RTIG) component. RTIG uses the Federate Object Model (FOM) that describes the classes of data exchanged by federates during execution. In [BS98], two security threats are identified in this architecture.

- Attacks on the communication links between the components or the RTI and
- Attacks via misuse or unauthorized use of RTI services.

The first threat is countered by creating a secure connection that ensures secrecy and authenticity of the data that is transmitted between the RTIA and RTIG. The second threat is countered by adding access control mechanisms within RTI services. Figure 7 shows the architecture of the system after having made provisions for countering the threats.

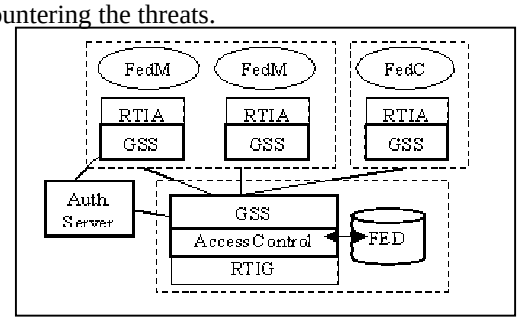

Figure 7. RTI security architecture, printed from [BS99]

To secure the communication between the RTIA and RTIG processes, GSS API [Lin97] is integrated within the RTIA and RTIG by extending the *socket* class that is used to exchange messages within RTI. GSS API (Generic Security Services Application Programming Interface) [Lin97] is a standard that defines the cryptographic services that can be used to secure a client-server application. Access control mechanisms are integrated within the RTIG and security labels are associated with federates and objects of the federation. [BS99] describes one more example of how security is integrated using a software architecture in systems using COTS based components.

Though this approach might look attractive at first, we believe it has a severe limitation- it requires the modification of the COTS components. According to us, the option of modifying the components themselves in order to incorporate security considerations is not a viable one because

- This inhibits the future reusability of the component
- In situations in which the source code of the component is not provided, it is not possible to modify the component at all.
- The effort involved in modifying the component might well exceed the effort that would be required to develop the component from scratch.
- Modification of components would entail regression testing [RH96] of the components. Without access to the source code such testing cannot be carried out faithfully.

- In some cases, legal restrictions against modifying their COTS components are imposed by the component manufacturers.

Apart from [BS99] and [BS98], very little work exists in this area. The limitations of the approach in [BS98] and [BS99] suggest that this is an area worth exploring by further research in Software Architecture. In the next section, we will look at a connector-oriented approach that might be a more pragmatic approach towards tackling such a problem.

3.5 Resolution of mismatched security-related interactions of heterogeneous software components

The problem of architectural mismatch [GAO95] is a major hindrance to the reuse of software. This problem occurs due to the mismatches in the assumptions of the reusable component about the system that it is to be a part of. In the context of security for a distributed software system with heterogeneous components, this problem manifests itself in the form of differing Quality of Service (QoS) requirements [BI97]. In [BI97], the authors Bidan and Issarny attempt a solution to the problem of grappling with complex security requirements of components in open distributed systems. Software Architecture is used to specify security requirements. After specification of the security requirements of each component, it is possible to build customized connectors (at compile time) that meet the security requirements of both the components involved in the connection. This is described in figure 7.

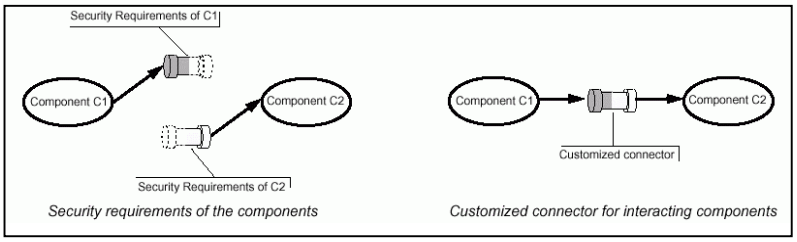

Figure 7. Security requirements and customized connectors, printed from [BI97]

The authors of [BI97] use this approach to specify the following categories of security requirements of components:

- Encryption requirements
- Authentication requirements
- Access control requirements

Each of these categories of requirements can widely vary from component to component. The challenge is to compare and compose these requirements so that heterogeneous components can talk to each other using the customized connector shown in figure g. We shall illustrate here how this challenge is addressed for the encryption requirements. Authentication and access control requirements are described in detail in [BI97].

Specifying the security properties related to encryption involves the specification of

- The encryption algorithm used
- The nature of parameters used by the encryption algorithm that is chosen

With heterogeneous components, mismatches can arise at either level- for the choice of the encryption algorithm as well as due to the nature of parameters used by the encryption algorithm. To handle these mismatches, the developer must first specify the different acceptable requirements for encryption as shown in figure g.

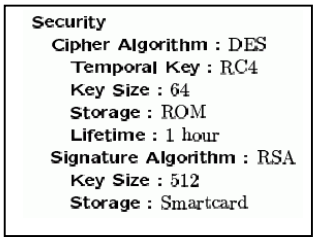

Figure g. Specifying security requirements, printed from [BI97]

Since each component can have a different set of encryption requirements like the one in figure g., it becomes necessary to compose and compare the encryption requirements. Since encryption and decryption functions are generally symmetric, a primary requirement for composition is that the encryption algorithm should be the same at both ends of a connector. This means that if we consider a connector as composed of two sub-connectors, one for each component, then the encryption algorithm should be the same for each of the two sub-connectors. In order to achieve the similarity of the encryption algorithm, the application developer is responsible for specifying a list of encryption algorithms in the two sub-connectors, along with the associated trust degree of the algorithm. Then, the encryption algorithm of the connector is that algorithm which belongs to both sub-connectors and has the highest trust degree. After the selection of the encryption algorithm, the parameters can be also selected according to this strategy. That is, a trust degree is associated with the parameters of the encryption algorithms and the parameters chosen are the ones that correspond to the highest trust degree and are specified in both the sub-connectors. A similar approach is adopted for the selection of authentication algorithms and access control mechanisms.

The above technique is a simple and effective way of resolving security-related architectural mismatch between components. An architecture-oriented approach is both intuitive and elegant and ensures that the security concerns are addressed early in the software development life cycle. This approach also lends itself well to resolving the security mismatches of legacy systems and components, a problem described in [DS00]. [FBF99] discusses a wrapper-based approach to the same problem.

In the next subsection, we shall have an overview of some benefits that can be accrued by the software developer by using principles of software architecture. The key distinction between the issues discussed in this section and those in the next section is that while the former focuses on architectural solutions to security problems, the latter essentially describes the benefits that can be obtained by developers of secure systems by using software architectures. These benefits may not necessarily contribute to enhance the security of the system.

3.6 Other Benefits obtained by using software architectures

The first benefit that can be obtained by using software architectures is the reuse of security-related code. It is an observation that security-related code of a software system does not generally lend itself well to reuse [JH98]. This is due to the fact that

- this code is typically embedded along with the functional code of the system and
- security-related code is typically specific to the particular system for which it was developed.

As described in [JH98] and [Dam98], abstracting security considerations to the level of software architecture by methods described in the section 3.5 enables the reuse of security-related code across varied applications. By managing the security considerations outside components, and in the connectors, the security and the functional aspects are made independent of each other, thereby facilitating reuse. As we shall see later, such separation of concerns can also be obtained using the Aspect Oriented Programming paradigm [KIL+97].

Architectural patterns are defined in [OR98] as

> *fundamental organizational descriptions of common top-level structure observed in a group of software systems.*

Architectural patterns capture and express earlier experiences in the design and development of software. In this manner, they provide a guideline for system developers during early stages of the software lifecycle. In [YB97], several architectural patterns for the security aspect of applications are presented. These patterns can be applied by the system developer both to serve as a security guideline for developers and reason about application security. Apart from [YB97], not much work has been done in this promising area. However, [BRD98] describes a pattern language for a generic object-oriented cryptographic architecture.

We believe that the use of software architecture concepts would also aid the developer in the development of security agents. According to [FL96], security agents are

> *ubiquitous, communicating, dynamically confederating agents that monitor and control communications among the components of preexisting applications.*

Software agents that implement security controls are described in [BGS+98] and [FL96]. The concept of software security agents as described in [FL96] is that agents can be implemented to monitor and perform access control, authentication, etc. by wrapping insecure components. These agents are called *SafeBots* in [FL96]. SafeBots have a certain level of intelligence associated with them in that they can adapt their actions to local and global context. Programming agents may be a promising step towards monitoring security of a distributed system. However, the programming of such agents is a very difficult task. This is due to the following reasons [TOH99]:

- Mobile agents might act in remote hosts with varied environments. This makes the task of predicting their behavior difficult.
- It is difficult to program agents that act in different platforms especially since the platforms themselves are being rapidly changed.

To counter these problems, in [TOH99], the authors propose an architecture oriented agent system development method based on agent patterns. A layered system architecture is defined in order to investigate a systematic agent development process. Furthermore, behavior patterns that correspond to the individual layers are devised in order to make the development of the layers easy. Behavior patterns are documentation of good past experiences in the development and behavior of agents. The layered agent system architecture is shown in figure 8.

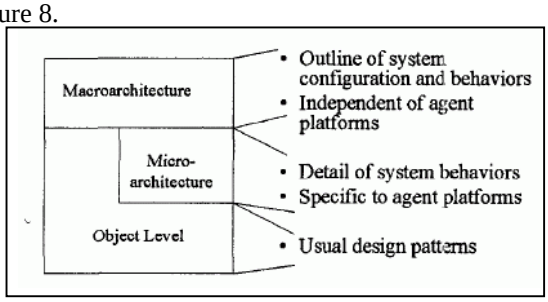

Figure 8. Layered agent system architecture, printed from [TOH99]

In the above figure, the macroarchitecture layer represents the outline of the system configuration. The macroarchitecture layer is independent of specific agent platforms. The microarchitecture layer describes the detail of system configuration and the agent behaviors specialized in each agent platform. The object level represents the implementation of the system depending on the design of the upper two layers. Behavior patterns that correspond to the individual layers described above are then determined. Finally, the three layers are designed in a top down fashion using these behavior patterns. Using such an architecture-based approach along with agent behavior patterns, it becomes easier for the developer to develop agent systems efficiently. The approach in [TOH99] is described for generic agents but it can be applied to the development of security agents. Related

work in the area of security agents can be found in [BGS+98] and [QS98].

3.7 Aspect Oriented Programming

The use of Software Architecture is one approach to constructing systems with evolutionary and reusable security. Another related approach, as identified in [DS00], is the use of the aspect oriented programming (AOP) paradigm [KIL+97]. The AOP paradigm explicitly provides for separation of concerns. In AOP, the different aspects of the system are programmed in their most natural form and then these different aspects are *woven* together to produce the executable code. Such an approach lends itself well to the separation of the security concerns (or the security *aspect*) from the functional features of the system. The AOP approach can be used to separate low-level security concerns as well high-level security concerns from the other concerns. While high-level security concerns refers to security risks that are external to the application (like intruders), low-level security concerns are those that refer to the application itself behaving in a insecure manner, possibly due to an attack (like buffer overflows).

In [VBC01], the authors Viega et al. describe the use of AOP to program security applications. They illustrate this by using an example of a language developed by them that extends the C programming language in order to support AOP. Such an extension can then be used to abstract security concerns outside the program proper. According to [VBC01], this would be helpful from the security point of view in a number of ways:

- Insecure function calls may be replaced by secure function calls.
- Buffer overflow can be prevented.
- Security audit trail and logging is possible.
- Generic socket code can be replaced by SSL socket code.
- Privileged sections of a program can be specified.

According to [DS00], the confluence of the field of software architecture and AOP is a promising prospective research area for engineering secure software.

4. SUMMARY

Very little work that specifically addresses the problem of engineering secure systems exists in the area of Software Engineering. In this paper, research in two seemingly independent areas- Software Engineering and Security- has been assimilated in order to

- demonstrate the solution of some problems in engineering secure systems and
- point to some research directions in the area of software engineering that would aid in the engineering of more secure systems.

With the understanding that principles of software architecture can be useful in solving many problems encountered in the development of distributed systems, we identified the work that has been done in the area of architecting secure systems. We categorized and abstracted certain common features among these approaches. We also identified certain problems with the existing approaches and presented some future research avenues that could be explored.

5. DISCUSSION

As pointed out earlier, the benefits that can be accrued by adopting a security-centered software engineering effort are enormous. However, we do not intend to imply that a software engineering effort should focus more on security than on other software qualities. But we do believe that security is not a quality that can be ‘pasted’ onto software once it is completed. If the software system that is being engineered is to be secure, security must be a concern throughout the software development life cycle. While efforts to attain software qualities like correctness and maintainability in a software product are being applied throughout the software development life cycle, security lags far behind in this respect. For example, a large amount of effort during software development does not contribute largely to the development task directly. A significant part of this effort is directed towards making the product more maintainable i.e. to induce the maintainability quality in the software. In fact, many software development organizations have begun to adopt specific tools and programming languages like Java in order to make their software more maintainable. This concern for software maintainability throughout the software development life cycle is justified, since if the software product is more maintainable, lesser maintenance costs will be incurred. In recent times, security attacks have also led to huge costs for the software user as well as the developer. Unfortunately, security has not received as much attention during software development as software maintainability has. We are not attempting to indicate that software organizations are careless about security. However, it is a known fact that not many software engineers have a formal background of security concepts. Also, a large impediment to the adoption of a security-centered approach to software development is the lack of adequate software engineering tools and techniques that help follow this security-centered approach. This may be attributed to the fact that the field of Software Engineering for Security is a relatively young field. However, we believe that in spite of a few limitations, the kind of work that has been surveyed in this paper is certainly a step in the right direction.

ACKNOWLEDGEMENTS

The author would like to thank Prof. David Rosenblum for the comments and suggestions provided by him during the course of writing this paper. The author would also like to thank Eric Dashofy for the discussion that led to the selection of this topic.

REFERENCES

[AG94] R. Allen, D. Garlan. Formalizing Architectural Connection. *Proceedings of the 16th International Conference on Software Engineering*. May 1994.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems, 8*. February 1990.

[BI97] C. Bidan, V. Issarny. Security benefits from software architecture. *Proceedings of COORDINATION'97: Coordination Languages and Models*. 1997.

[BP75] D. E. Bell, L. J. LaPadula. Secure Computer system: Unified Exposition and Multics Interpretation. *Technical Report MTR-2997, MITRE Corporation, Bedford, MA*. July 1975.

[BRD98] A. M. Braga, C. M. F. Rubira, R. Dahab. Tropyc: A Pattern Language for Cryptographic Software. *In 5th Pattern Languages of Programming (PLoP'98) Conference*, 1998.

[Bro87] F. P. Brooks, Jr. No Silver Bullet; Essence and Accidents of Software Engineering. *IEEE Computer 20(4)*. April 1987.

[BS98] P. Bieber, P. Siron. Design and Implementation of a Distributed Interactive Simulation Security Architecture. *Proceedings of the 3rd International Workshop on Distributed Interactive Simulation and Real-Time Applications*. 1998.

[BS99] P. Bieber, P. Siron. Security Architectures for COTS Based Distributed Systems. Available at http://www.cert.fr/francais/deri/bieber/papers/ist11/

[BSW01] I. Bashir, E. Serafini, K. Wall. Securing Network Software Applications: Introduction. Communications of the ACM 44, 2. February 2001.

[CB94] B. Cheswick, S. Bellovin. Firewalls and Internet Security: Repelling the Wily Hacker. *Addison-Wesley*. 1994.

[Chi98] D. Chizmadia, A Quick Tour Of the CORBA Security Service. *Information Security Bulletin*. September 1998.

[CN95] L. Chung, B. A. Nixon, "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach." *Proeedings of the 17th ICSE, Seattle, WA, U.S.A.* April 1995.

[Dam98] Christian Damsgaard. Secure Software Architecture. *Jensen Project SIRAC, IMAGINRIA 655. Avaliable at citeseer.nj.nec.com/10161.html*. 1998.

[DGS99] P. Devanbu , M. Gertz , S. Stubblebine. Security for Automated, Distributed Configuration Management. *In Proceedings, ICSE 99 Workshop on Software Engineering over the Internet*. 1999.

[DS99] P. Devanbu, S. Stubblebine. Cryptographic Verification of Test Coverage Claims. *IEEE Transactions on Software Engineering*. 1999.

[DS00] P. Devanbu, S. Stubblebine; Software Engineering for Security: A Roadmap. *Proceedings of the conference on The future of Software engineering.* 2000.

[DWW99] A. Dima, J. Wack, S. Wakid. Raising the Bar on Software Security Testing. *IT Professional.* Vol. 1, No. 3, May/June 1999.

[FBF99] T. Fraser, L. Badger, M. Feldman. Hardening COTS Software with Generic Software Wrappers. *IEEE symposium on Security and Privacy*. 1999.

[FKA+94] G. Fink, C. Ko, M. Archer, K. Levitt. Toward a Property-based Testing Environment with Application to Security Critical Software. *Proc. of the 4th Irvine Software Symposium*. April 1994.

[FL96] R. Filman, T. Linden, SafeBots: a Paradigm for Software Security Controls. *Proceedings of New Security Paradigms Workshop, Lake Arrowhead, CA USA*. 1996.

[GAO95] D. Garlan, R. Allen and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. *Proceedings, 17th International Conference on Software Engineering, Seattle WA*. April 1995.

[Gas88] M. Gasser. Building a Secure Computer System. *Van Nostrand Reinhold - New York, NY*. 1988.

[BGS+98] J. Balasubramaniyan, J. O. Garcia-Fernandez, E. H. Spafford, D. Zamboni. An Architecture for Intrusion Detection using Autonomous Agents. *Department of Computer Sciences, Purdue University; Coast TR 98-05*. 1998

[GM82] J.A. Goguen and J. Meseguer. Security Policy and Security Models. *Proceedings of the 1982 IEEE Symposium on Research on Security and Privacy, IEEE Press*. 1982.

[Gon97] L. Gong. New security architectural directions for Java. *Proceedings of COMPCON '97*. 1997.

[GS01] A. K. Ghosh, T. M. Swaminatha. Software Security and Provacy Risks in Mobile E-commerce. *Communications of the ACM* 44, 2. February 2001.

[FL94] G. Fink, K. Levitt. Property-based testing of privileged programs. In Tenth Annual Computer Security

Applications Conference, pages 154--163. *IEEE Computer Society Press*. December 1994.

[HP87] A. Herzberg, S. S. Pinter. Public Protection of Software. *ACM Transactions on Computer Systems, 5(4):371—393*. November 1987.

[HW96] D. Heimbigner, A. L. Wolf. Post-Deployment Configuration Management. *Software Configuration Management: ICSE'96 SCM-6 Workshop Selected Papers (Berlin, Germany)*. 1996.

[JH98] C. Jensen, D. Hagimont Protection Reconfiguration for Reusable Software Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98). 1998.

[Kem89] R. A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications, 7*. 1989.

[KIL+97] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. V. Lopes, C. Maeda, A. Mendhekar. Aspect-Oriented Programming. *A Position Paper from Xerox PARC*. 1997.

[Lin97] J. Linn. Generic Security Service Application Programming Interface. *Internet RFC 2078*. January 1997.

[LSS+80] R. Locasso, J. Scheid, D. V. Schorre, P. R. Eggert. The Ina Jo Reference Manual. *Technical Report TM-(L)-6021/001/000, System Development Corporation, Santa Monica, California*. 1980.

[LV95] D.C. Luckham, J. Vera. An Event Based Architecture Definition Language. IEEE Transactions on Software Engineering Vol. 21, No 9, September 1995.

[McL94] J. McLean. Security Models. *J. Marciniak, editor, Encyclopedia of Software Engineering. Wiley Press*. 1994.

[MQR+97] M. Moriconi, X. Qian, R. Riemenschneider, and L. Gong. Secure Software Architecture. *Proceedings of the IEEE Symposium on Security and Privacy*. 1997.

[ML97] S. Meldal and D. Luckham. Defining a Security Reference Architecture. *Technical Report CSL-97-728 Program Analysis and Verification Group Report No. 76 Computer Systems Laboratory Stanford University*. 1997.

[MM84] T. Maude, D. Maude. Hardware protection against software piracy. *Communications of the ACM, 27(9)*. September 1984.

[MV99] G. McGraw, J. Viega. Why COTS Software Increases Security Risks. *ICSE Workshop on Testing Distributed Component-Based Systems*. May 1999.

[OR98] J. Ortega-Arjona, G. Roberts. Architectural Patterns for Parallel Programming. *Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing*. 1998.

[PW92] D. E. Perry, A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes, 17*. October 1992.

[QS98] Q. He, K. P. Sycara. Towards a Secure Agent Society. *ACM AA'98 Workshop on Deception, Fraud and Trust in Agent Societies*. 1998.

[RH96] G. Rothermel, M. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering, 22(8)*. August 1996.

[RSC97] Rational Software Corporation. The Unified Modeling Language. *Documentation Set Version 1.0, Santa Clara, CA*. January 1997.

[Sch96] B. Schneier. Applied Cryptography, Protocols, Algorithms, and Source Code in C. *Second Edition, John Willey & Sons*. 1996.

[TOH99] Y. Tahara, A. Ohsuga, S. Honiden. Agent System Development Method Based on Agent Patterns *Proceedings of The Fourth International Symposium on Autonomous Decentralized Systems*. 1999.

[VBC01] J. Viega, J.T. Bloch, P. Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*. February, 2001.

[VKP01] J. Viega, T. Knono, B. Potter. Trust (and MisTrust) in Secure Applications. *Communications of the ACM*. February 2001.

[WBD+97] D. S. Wallach, D. Balfanz, D. Dean, E. W. Felten. Extensible Security Architectures for Java. *Technical Report 546-97, Department of Computer Science, Princeton University*. April 1997.

[WWW1] http://www.whatis.com

[WWW2] OMG. The Common Object Request Broker Architecture (CORBA). http://www.omg.org/

[YB97] J. W. Yoder and J. Barcalow. Architectural Patterns for Enabling Application Security. *In Proc. 4th Pattern Languages of Programming, Monticello, IL, September 1997. Available as Washington University Technical Report WUCS--97—34*. 1997.