Proceedings of the 2004 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY 10-11 June

SECURITY IN SOFTWARE ARCHITECTURE: A CASE STUDY

Adam Sachitano, Richard O. Chapman, Ph.D., *Member, IEEE* and John A. Hamilton, Jr.,Ph.D., *Senior Member, IEEE*

Abstract – ***The idea of ensuring that non-functional requirements such as maintainability, performance, reusability, reliability, and others are designed and built in to new software is not a new one. However, software security as a particular non-functional requirement of software systems is all too often addressed late in the software development process. As a result, the security of such software systems is poor and can lead to security compromises (such as theft of service or information), increased costs in maintaining the software, and many indirect costs such as system downtime, loss of productivity, etc. This paper will survey existing research into architecting and engineering security and will present this material along with a case study of Daniel J. Bernstein's qmail Internet mail server software.***

Index terms – Software Architecture, Security

I. INTRODUCTION

Non-functional requirements, or NFRs, are software requirements that are separate from procedural requirements specified in a requirements document. These include somewhat subjective requirements such as maintainability, performance, reusability, reliability, security, and others. There is often no exact specification for these requirements, nor are there metrics for specifying objective tolerances of these requirements. In short, functional requirements designate *what* a piece of software should do, but NFRs describe *how* that software should perform those tasks.

NFRs are often of extreme importance to a software system, in spite of their often-secondary treatment in the software development process in favor of functional requirements. In addition, software architecture is becoming an increasingly important and studied topic in software engineering, and many have asserted the importance of addressing such non-functional concerns in software architectures [1].

The focus of this paper will be on the NFR of security, and will present the area of Software engineering known as security engineering, along with justifications for its inclusion in modern university degree programs. Next, this paper will address a few of the proposed means for security engineering in the software lifecycle model, and will continue with a discussion of security at the architecture level.

II. SECURITY ENGINEERING

Some basic requirements for building secure systems include an in-depth examination of current practices, as well as instruction in correct methodology of the construction of future systems. One of the tenets found in much of the literature reviewed for this paper is common to ensuring other NFRs: software security needs to be considered from the very beginning of the software development lifecycle [2],[3][4]. In general, when a software system is constructed without a great deal of forethought into security as an NFR, security considerations are addressed afterwards. One reason for this is that product development by necessity must optimize the use of limited resources (time, funds, personnel, etc.), and yet in order to be competitive must utilize these resources to deliver the most value to customers as early as possible, leaving security concerns for later consideration [3]. This can lead to design-level security flaws because the primary goal of the developers was developing a functional system, rather than a functional and secure one. Exploitation of security flaws in such software often occurs due to *circumvention* of security mechanisms, rather than *breaking* such mechanisms [4]. A software system that has not been constructed from beginning to end without adequate attention to how such circumvention can be avoided is more apt to such flaws.

In order to ensure adequate attention in all appropriate stages of the software development lifecycle, security engineering is needed as a necessary part of software engineering. "Security engineering is about building systems that are and can remain dependable in the face of malice, error or mischance. As a discipline, security engineering focuses on the tools, processes and methods needed to design, implement and test complete systems, and to adapt existing systems as their environment evolves [5]." This requires, almost by definition, an attention to security even at the beginning of the software system's lifecycle model. It can be very difficult to, for example, design and implement a communication

0-7803-8572-1/04/$20.00 ©2004 IEEE

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on May 24,2025 at 18:10:23 UTC from IEEE Xplore. Restrictions apply.

protocol in a software system, and then address security considerations such as encryption afterwards. Software design and product engineering that doesn't address this security concern from the start may leave the software open to a security flaw due to the system's fundamental design or construction.

In addition, "security engineering requires cross-disciplinary expertise, ranging from cryptography and computer security, through hardware tamper-resistance and formal methods, to knowledge of applied psychology, organizational methods, audit and the law. System engineering skills - from business process analysis through software engineering to evaluation and testing - are also important, but they are not sufficient. They only deal with error and mischance rather than with malice [5]." In other words, it is not enough for the software engineer to have made the design decision that, for example, "this communication channel must be secure." The engineer must know how to properly implement security measures which guarantee security. This requires more than just a passing knowledge in, as Anderson says, cryptography and computer security.

Security engineering must, by necessity, become a more important aspect of software engineering as a whole. The trend towards so-called 'ubiquitous computing' that many predict dictates that more critical systems must be secure while being more exposed to everyday use. As Anderson states, "The last twenty years have seen much work on the theoretical aspects of computer security and cryptology. But there has been much less on the practice. Many insecure systems are built, and the resulting safety, privacy and crime prevention problems [...] are a significant impediment... Once communicating embedded systems become both ubiquitous and critical, we will simply have to do better [5]."

Software engineering curricula that incorporate security engineering into a student's training is endorsed by [6]. Increased academic interest in security is evidenced in the emergence of computer science and software engineering curricula in many institutions that offer concentrations in information security. Examples include the Naval Postgraduate School Center for Information Systems Security Studies and Research (NPS CISR), George Mason's Laboratory for Information Security Technology, and the James Madison University concentration in Information Security Master's degree program [6]. As described above, Vaughn asserts, "The topic of security in computing requires a foundation of study in operating systems, database systems, networks, architectures, and ... artificial intelligence." Vaughn suggests that a comprehensive approach in CS curricula including the addressing of security concerns in each of these areas along with a final, senior-level course focusing specifically on information security issues is preferred.

Many academic institutions are beginning to offer Software Engineering degree programs, including Mississippi State University and that of the authors Auburn University. It is important to note that these programs are more directed at the practical instead of the theoretical. Vaughn emphasizes that the competent software engineering graduate should view computers and programming languages as tools with which practical solutions to a given problem may be implemented in a fashion that is, along with other software engineering considerations, secure. In the closing of his discussion of Software Engineering degree programs, Vaughn states, "It is the opinion of this author that where software engineering programs exist, information security courses should be a requirement. Security is ... a user requirement that must be satisfied. The [software engineering] student must be trained in how to meet this requirement... [6]."

As noted above, security as an NFR is often considered as an after thought to some portion (large or small) of the beginning of a software product's life cycle. Of course, there are cases when this is not an option for a software developer. For example, the advent of networking and open standards often provide new business reasons to re-engineer legacy systems (which may have operated within trusted intra-nets) for operation over a public network such as the Internet [3]. In such cases, there is no alternative but to consider security after initial design and development. One of the specific issues addressed by Devanbu and Stubblebine is when re-engineering such a legacy system causes developers to have to change security concerns from a security model such as Unix to a framework such as CORBA. This is referred to as a legacy security mismatch, as these two platforms have different security policies and enforcement mechanisms [3].

Another case that partially falls into this category is having to design a system using common off-the-shelf (COTS) components. It is a common theme in the literature that using such components in safety- or security-critical applications is risky [7], [3]. Evaluating black-box components for concerns such as security, safety, and fault-tolerance is difficult, as the requirements of a possible customer of such components are often at odds with component vendors who may risk intellectual property loss. Devanbu and Stubblebine present what they call a grey-box approach to COTS component verification, while Guerra, et. al., advocate an approach (specifically targeted at fault-tolerant systems) that employs a C2-based architectural style for structuring the addition of error detection and recovery mechanisms.

0-7803-8572-1/04/$20.00 ©2004 IEEE

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on May 24,2025 at 18:10:23 UTC from IEEE Xplore. Restrictions apply.

Such mechanisms are to be added to the component during system integration.

III. Addressing Security in Software Design and Modeling

The Unified Modeling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a "blueprint" for construction [8]. In [4], the author presents a method of secure system development that includes the use of goal trees, formal semantics, and an extension to the UML called UMLsec. In his work, Jürjens asserts that while functional requirements are analyzed carefully in system design and development, security considerations often arise "after the fact." Also, he states, "the relationship between non-functional requirements and software architectures is only very poorly understood [4]." Jürjens suggests improvements to this situation by integrating security requirements analysis with a standard development process, and modeling of security related features such as confidentiality, access control, etc. through UMLsec. UML was chosen for extension in order to make it available to developers who may not be specialized in security, but who have experience in software modeling with UML.

UMLsec consists of standard UML diagrams including: use case diagrams, activity diagrams, class diagrams, system sequence diagrams, state diagrams, and deployment diagrams. UMLsec is an extension in typical UML fashion: via labels, such as stereotypes, and key-value pairs. System sequence diagrams are naturally useful for specifying security protocols, and state diagrams are useful for modeling dynamic, event-driven system behavior in which events that meet certain system conditions (such as whether a correct response to a challenge has been given) cause transitions from one state to another. Deployment diagrams are used to describe the physical layer, helping the developer to ensure that security requirements on communication are met by the physical layer. An example from the realm of on-line commerce is used to generate each of these diagrams, and an explanation of each is provided.

Jürjens has extended UML to encapsulate knowledge on prudent security engineering and thus has attempted to make it available to developers not specialized in security. The intent is that UMLsec modeling should highlight aspects of a system design that can give rise to vulnerabilities. A UMLsec model that has been developed early in the software life cycle should not only help the developers evaluate their design for potential vulnerabilities, but should also give a more complete model from which to translate security design concerns directly to implementation.

A math object is defined as any equation or fragment containing mathematical symbols (including Greek characters, superscripts, and subscripts) that appears either in-line (in the flow of normal text) or as a display equation (in its own space between lines of text). In particular, you must avoid using Word fonts or character calls for in-line "quickies" such as single variables with superscripts or subscripts.

IV. System Design Based on Security Architecture

Schneider [9] defines different architectures (e.g., security, system, and software architectures) that together form an overall architecture that satisfies the constraints of each of its constituent architectures. Schneider asserts that a system's design is the product of these architectures. This paper is based on concepts from the Defense Goal Security Architecture (DGSA), which is an architectural framework in which system architects define security according to the requirements to protect information [9], [10]. In other words, the basis for such an architecture is independent of everything but the information one must protect.

Schneider defines architecture as "[providing] a structure through which a large or complex system can be understood and reasoned about. Weaknesses can be identified before the system is built [9]." The assertion that architectures can be evaluated before the construction of a software system has been presented in various different ways (e.g., [11]; [12]). Like Jurjen's work with UMLsec mentioned above, the idea is to consider security early on in a system's design with the intent of producing a more secure end product. The products of this design phase (a security architecture or a UMLsec model) are to be developed early, and evaluated as far as those artifacts can be evaluated before they are implemented.

Schneider discusses security architectures in the context of information domains and their connections to one another (these concepts are all rooted in terminology set forth in the DGSA). Information domains partition information in a system so that all information in a particular domain has the same protection requirements. A set of users that has access to a particular domain shares all of the information in a domain. The transfer of information from one domain to another forms a connection between components. A user can make a transfer if the system security policy allows it to share information in both the source and destination information domains [9]. An example is then given in the domain of a university system in which one professor shares grades and other sensitive information (such as

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on May 24,2025 at 18:10:23 UTC from IEEE Xplore. Restrictions apply.

attendance records, etc.) with the corresponding student (and only that student), and limited information about all grades (such as distribution) is available to anyone. In this example, the professor has access to all of his students' grades (but not another professor's), while a student only has access to his own grades (regardless of professor). The distribution of grades is available to anyone, though the information by which that distribution was calculated (the individual grades) is not. Therefore each professor comprises one information domain, and each student has a separate information domain as well as being part of the unrestricted global information domain. The connections logically follow from the description above.

Information domains are, by definition, independent of software architecture. A user may need simultaneous access to information of different types, each managed by a different server; thus, different components of the software architecture (such as grades or attendance records in the above example) may coexist in an information domain. As mentioned before, an overall architecture may be composed of component architectures that are each separate. This places such an architectural view of a system conceptually above a software model such as one that might be developed in UMLsec, as the different views possible in a UMLsec model are combinations of different architectures. For example, a system sequence diagram in UMLsec may be the combination of both software and security concerns (Schneider's software and security architectures, respectively), while a deployment diagram is the combination of software, security, and hardware concerns (Schneider's software, security, and system architectures, respectively).

Schneider submits that an overall architecture is a three-dimensional product, of which security is one of the independent dimensions (the other two being end systems and software managers, corresponding to system and software architectures, respectively). Schneider does not describe how these three architectures are to be translated into an implementation. Some guidelines are given; for example, "a computation on a system constructed using the concepts presented here occurs in a domain containing the minimal amount of information necessary." However, Schneider himself states, "Current commercial systems do not provide necessary support for architectures developed using the methods described in this paper. The required isolation has not been a design goal for current systems, and consequently most are inadequate." So not only is a method for translation not presented, but apparently current platforms are inadequate to provide for the basic security architectures he presents. Indeed, [10] acknowledge that no system implementation conforming to the DGSA has been produced, although progress is being made to that end. With that being said, the high-level concepts set forth in this field of research are valuable when considering security in a high-level architecture. Though a systematic method of translating such an architecture into an implementation has not been proposed, one could argue that there is neither an exacting and systematic method of translating any architecture into a software implementation.

V. Introduction to qmail Case Study

One of the common concerns brought up in a graduate software architecture class that the author has attended is that no existing software project architecture has been such an overwhelming success (due to its initial design and architecture considerations) that it is used as a pedagogical example with which students are instructed in the art of proper software architecture development. Through personal interests and research, the author has identified the qmail software project as potentially being one such project which, given its design considerations, shows such a noteworthy feat of software architecture and engineering.

qmail is a modern UNIX-targeted SMTP server written by Daniel Bernstein as a replacement to the sendmail SMTP server software project [13]. A simple keyword search for 'sendmail' on the CERT (a computer security incident response team) advisories, incident notes, and vulnerability notes database returns 126 documents, 38 of which are advisories. Over half of the sendmail advisories are remotely exploitable security flaws [14]. Daniel Bernstein wrote the qmail server software, starting in January 1996, as a reaction to these flaws [13]; [15]. According to Bernstein, as of October 2001, more than 700,000 reachable IP addresses on the Internet were running the qmail mail server software [16].

VI. qmail Design Considerations and Software Architecture

To start with, qmail's architecture is one of 'pipe-and-filter,' in which many programs are 'strung together' to complete a task. Each program handles a certain task, and the output of that program is 'piped' as the input to the next program down the line. As such, a qmail SMTP server is composed of many different programs which all handle a certain task in the overall services of sending and receiving mail.

qmail is an example of a software project that was conceived, designed, and implemented with security in mind from the very beginning. Bernstein (in [15] states that he followed seven rules in the design and implementation of qmail:

1. Programs and files are not addresses

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on May 24,2025 at 18:10:23 UTC from IEEE Xplore. Restrictions apply.

2. Do as little as possible in setuid programs
3. Do as little as possible as root
4. Move separate functions into mutually untrusting programs
5. Don't parse
6. Keep it simple
7. Write bug-free code

Rule 1 is sendmail specific, as sendmail treats programs and files as addresses as part of one of its design decisions, the result of which is that certain security restrictions can be circumvented allowing random users to execute arbitrary programs or write to arbitrary files. Rules 2, 3, and 4 play into security design and architecture decisions that will be discussed later. Rules 5 and 6 are design decisions that will briefly be explained later. Finally, lest the reader scoff at rule 7, Bernstein does not assert that his code is bug-free. The this rule appears to be a jab at the C standard library routines (such as those relating to stdio), which are, according to Bernstein, designed to encourage bugs (through programmer carelessness).

Rule 2 is a design and architecture decision made by Bernstein almost certainly through common sense and as a reaction to many vulnerabilities in sendmail. In UNIX, setuid programs run with the effective userid of the root user (or superuser). They are vulnerable to the original invoking user taking control of the process's file descriptors, environment, terminals, timers, signals, and other items. All of these could conceivably be used to trick a setuid program into, among other things, escalate a user's privileges. Only one qmail program is setuid. Its function is only to add a new mail message to the outgoing queue [15].

Rule 3 is another design and architecture decision made by Bernstein for much the same reasons as rule 2. When a program runs as root, there's no way that its mistakes can be caught by the operating system's built-in protections. The concept at work in both rules 2 and 3 is that of reducing privileged actions to the smallest scope of execution possible. Only two qmail programs run as root, as opposed to the entire sendmail system [15].

Rule 4 is another design and architecture decision along the lines of rules 2 and 3. Different programs composing a qmail SMTP server run as one of 3 different users, and the programs in one group do not trust the programs in another. Even if an attacker were to compromise 3 of these programs and gain control of these 3 user accounts, he still could not gain control of one's system. From a privilege escalation standpoint, as long as root does not send mail, none of these programs will be invoked with root's userid, and the only programs that could be considered security-critical do not write any files nor do they start other processes as root [15].

Rule 5 is a design consideration made by Bernstein. Parsing is, according to Bernstein, converting an unstructured sequence of commands (in a format determined more by psychology than by solid engineering) into structured data. Bernstein shuns parsing as a recipe for disaster. Some inputs may not be handled correctly, and some outputs may be produced which do not have the correct meaning. Apart from being a good rule of thumb in security engineering, it is common sense to see that when a malicious user controls the original data, such bugs can translate into security holes. Command-line parsing is not done (this is part of the reason why qmail has so many different programs), and all internal file structures are designed in a way that attempts to avoid confusion. Those RFC-822 features that require parsing of address lists and rewriting email message headers is handled by a program which runs without privileges [15].

Rule 6 is also a design consideration made by Bernstein. One of the many beneficial side effects of the pipe-and-filter architecture is that new functionality can be added by altering or redirecting input or output into custom programs. In effect, 'plug-ins' can be written by third parties, and run with qmail at one's own risk. This risk can be mitigated if the developers of such plug-ins understand qmail's security-based design considerations and develop their plug-ins accordingly.

As explained briefly above, rule 7 is Bernstein's echo of a lament shared by many who detest the C programming language. The C standard library of memory handling and i/o handling functions are, when used by inexperienced or careless programmers, prone to bugs such as buffer overflows due to (for example) format-string attacks on the printf family of functions. For the purposes of his software, Bernstein uses replacements for these functions that he has designed and implemented himself [15].

VII. QMAIL'S SUCCESS

As stated before, qmail is a software project which, given its design considerations, is such a noteworthy feat of software architecture and engineering that it could be used as an example from which to instruct future software engineers in the art of software architecture and security engineering. qmail's design considerations included security from the very start, and this is evident in that a keyword search of the CERT database for 'qmail' results in zero advisories, vulnerability notes, or incident notes [14]. Nobody has ever found any security holes in qmail – ever [15]. In fact, starting in March 1997, a cash reward has stood for anyone who identifies a security hole in qmail. The prize has so far gone unclaimed.

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on May 24,2025 at 18:10:23 UTC from IEEE Xplore. Restrictions apply.

qmail is not just secure. It is also a very successful SMTP server for its relative youth. A survey conducted by Bernstein shows that qmail is the fastest-growing SMTP mail server software on the Internet with, at last survey, an increase from 9 to 17% of responding mail servers. Sendmail servers declined from 47 to 42% of responding mail servers in the same period. In addition, as of the latest 2001 survey, qmail was only slightly behind Microsoft Exchange/IIS SMTP servers [17].

A number of large Internet sites currently use qmail, including: Yahoo! mail, Network Solutions, Verio, MessageLabs (searches 100M emails/week for malware), listserv.acsu.buffalo.edu (a big listserv hub, using qmail since 1996), Ohio State (the largest US University), Yahoo! Groups, USWest.net (Western US ISP), Telenordia, gmx.de (German ISP), NetZero (free US ISP), Critical Path (email outsourcing service w/ 15M mailboxes), and PayPal/Confinity [18]. qmail's ability to handle stresses such as those it is subjected to at the above-mentioned sites gives testament to its stability and reliability. The author uses qmail on a small, family mail server as well as a small business's mail server. The software is very stable once up and running, and has never been anything but easy to deal with.

It is for these reasons that the authors believe that qmail may very well be a software project which could serve as an example of a 'good,' secure, and successful architecture. Security considerations such as limiting the scope of privileged operations and separation of privileges implemented within the context of a pipe-and-filter architecture make qmail a successful, secure software architecture, given its design goals.

In this paper, a survey of the field of security engineering is presented, along with justifications for the necessity of including such training into software engineering curricula. Relatively current research into the field of security as it pertains to software engineering, modeling, and architecture was presented. Finally, a review of the qmail Internet SMTP server software was presented.

It is obvious that in order to further the field of software security, security must play an integral role in the training of software developers. In particular, software engineers must be conditioned to consider the security of their software products from the early stages of architecture and design. Architectures must be formulated with security in mind, and a skilled developer or designer must be able to translate these architectures and requirements into early software design artifacts that include some notion of security, at least in principle, if not down to detailed descriptions of necessary security mechanisms. The developer must then take advantage of this preliminary work in the early software life cycle model and implement that system with security in mind during all phases of development, testing, and maintenance.

VIII. References

[1] Chung, L; B. A. Nixon; and E. Yu, *An Approach to Building Quality into Software Architecture*, Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research, p 13, IBM Press 1995.

[2] Wang, H, and C. Wang, *Taxonomy of Security Considerations and Software Quality*, Communications of the ACM, Vol 46, Issue 6, p 75-78, ACM Press 2003.

[3] Devanbu, P. T.; S. Stubblebine, *Software Engineering for Security: a Roadmap*, Proceedings of the Conference on The Future of Software Engineering, p 227-239, ACM Press 2000.

[4] Jürjens, J., *Using UMLsec and goal trees for secure systems development*, Proceedings of the 2002 ACM symposium on Applied computing, p 1024-1030, ACM Press, 2002.

[5] Anderson, R., Security Engineering: A Guide to Building Dependable Distributed Systems, Wiley 2001.

[6] Vaughn, R. B., Jr., *Application of Security to the Computing Science Classroom*, Proceedings of the thirty-first SIGCSE technical symposium on Computer Science Education, p 90-94, ACM Press 2000.

[7] Guerra, P. A., C. M. F. Rubira, A. Romanovsky, and R. de Lemos, *A Fault-Tolerant Software Architecture for COTS-Based Software Systems*, Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, p. 375-378, ACM Press 2003.

[8] UML Resource Center, 2003, (November 30, 2003) http://www.rational.com/uml/index.jsp.

[9] Schneider, E. A., *Security Architecture-Based System Design*, Proceedings of the 1999 Workshop on New Security Paradigms, p 25-31, ACM Press 1999.

[10] Feustel, E. A. and T. Mayfield, *Unmet Information Security Challenges for Operating System Designers*, ACM SIGOPS Operating System Review, p 3-22, ACM Press 1998.

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on May 24,2025 at 18:10:23 UTC from IEEE Xplore. Restrictions apply.

[11] Lung, C.; S. Bot; K. Kalaichelvan; and R. Kazman, *An Approach to Software Architecture Analysis for Evolution and Reusability*, Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, p 15, IBM Press 1997.

[12] Gomaa, H; D. A. Menascé, *Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures*, Proceedings of the Second International Workshop on Software and Performance, p 117-126, ACM Press 2000.

[13] Sill, D, 2003, *Life With qmail,* (Dec 1, 2003) http://www.lifewithqmail.org/lwq.html.

[14] CERT, 2003, *CERT Coordination Center Homepage*, (Dec 1, 2003) http://www.cert.org.

[15] Bernstein, D, 2001, *The qmail Security Guarantee*, (Dec 1, 2003) http://cr.yp.to/qmail/guarantee.html.

[16] Bernstein, D, 2001, *qmail FAQ,* (Dec 1, 2003) http://cr.yp.to/qmail/faq/orientation.html.

[17] Bernstein, D, 2001, *SMTP Survey 20011003*, (Dec 1, 2003) http://cr.yp.to/surveys/smtpsoftware6.txt.

[18] Nelson, R, 2003, *qmail homepage,* (Dec 1, 2003) http://www.qmail.org/top.html.

Authorized licensed use limited to: Vrije Universiteit Amsterdam. Downloaded on May 24,2025 at 18:10:23 UTC from IEEE Xplore. Restrictions apply.