Principles and Patterns for Security

Andrew M. Schwerin
schwerin@cs.washington.edu

February 25, 2002

1 Introduction

One reason that security is such a challenging topic in software engineering is that its meaning varies widely. Its exact meaning depends upon the specific application being developed, and even upon the specific part of the application under development. It includes data confidentiality, data integrity, user authentication, authorization and data availability [13]. Secret passwords and sophisticated encryption algorithms are some of the tools available to help provide security, but they are not the only ones, and they are not security in and of themselves.

The process of creating and deploying a secure application places burdens on application designers, developers and maintainers, and on the users who requisition the software. The users must have a security policy. That is, they must have or develop an explicit understanding of the value (monetary, personal, etc.) of different kinds of data. Designers and developers must understand the security policy and the application domain well enough to design enforcement of the policy into the system. Maintainers must understand the policy and how the system enforces it in order to avoid compromising the security of a deployed system when the policy changes [12].

In this paper, existing design patterns and principles for the development of secure applications will be examined. Some of the major design patterns developed over the last eight years will be discussed, as well as some of the key principles that have emerged in the design of secure systems. Research into the use of Virtual Machines to provide security for mobile code is highlighted briefly, and suggestions for future work are made. For a good review of

the concept of design patterns and for a catalog of common software design patterns, the reader is referred to [6], the excellent book by Gamma *et al.*

2 Security

To begin with, an extremely cursory review of some security terms is in order. More detailed explanations can be found in [11], [13] and sources cited in those documents.

2.1 Confidentiality and Integrity

One common goal of a security policy is to keep certain data confidential. That is, to prevent read access to the data by entities not authorized to do so. An email serving application would have this goal. Another common goal is to guarantee the integrity of certain data. In the words of the Secure Linux Programming HOWTO, maintaining integrity means ensuring that "assets can only be modified by authorized parties in authorized ways [13]." For example, most uses of web serving systems restrict modification, but not reading of data. Both goals are achieved through the proper use of cryptography, along with the proper logical and physical separation of data from unauthorized users. How data are protected should depend upon the costs of unauthorized access and manipulation [12].

2.2 Authorization

Since private data should be accessible only to certain principals (parties in a secure application), the authorization of those principals is an important part of the design of a secure system. Authorization is the process of choosing to allow a principal to take action on some data. As an example, users logged into a Unix system control which other users are authorized to access their files by setting file permissions. When users attempt to access files, the operating system verifies their authorization to do so by comparing group and user ids and checking the file permission attributes.

2.3 Authentication

The process of authentication is that of identifying a principal. In the physical world, this is the role of passports, driver's licenses and other identification cards and tokens. In computer systems, it often involves passwords, but could also involve proofs of cryptographic key possession, biometric scans, or other identifying proofs. The process is distinct from that of authorization, and one may exist independently from the other.

2.4 Availability

Surprisingly to some, it is not sufficient to define a secure system as one that prevents unauthorized users from accessing data. If this were enough, unplugging the network cable and locking the computer up in a well-secured room would suffice. Complicated protocols for authentication and authorization could be replaced with a very heavy door and a guard. Users whom the guard let into the room would know the data was valid, because they'd be getting it directly from the source. The guard's sour disposition would ensure the requisite privacy.

However, since most data is only useful if it is accessible, it is necessary to to do more than secure data against unauthorized access and manipulation. It is necessary to design systems so that adversaries cannot easily reduce availability of data. The more important timely access to the data is to legitimate users, the more costly attacks that result in denial of service will be. Because of this, availability is a major security concern.

3 Security Design Patterns

The first principle identified by the NIST publication addressing the engineering principles underlying information technology security is to "establish a sound security policy as the 'foundation' for design [12]." This principle is generally accepted throughout the academic and professional literature (see, e.g., [15], [9], [13]). One implication of this principle is that security cannot only be addressed in the design and implementation of an application. Part of the specification process must identify the security goals explicitly, so that the application

may be designed with them in mind, and so that the application may be tested with respect to them. So that the users do not become the weakest link in the security chain, it is important that the users understand the security risks and associated protection measures involved in an application.

In [15], Yoder and Baraclow discuss several object-oriented software design patterns that are useful when designing applications both before and after the security policy for the application is fully specified. These design patterns are primarily targeted at the application developers, but do not provide guidance on the creation of a security policy. The patterns discussed are *Single Access Point*, *Check Point*, *Roles*, *Session*, *Secure Access Layer*, and two patterns that deal primarily with the view of the system presented to the user, known as *Limited View* and *Full View with Errors*.

The *Single Access Point* pattern restricts access to an application to a unified entry point. Yoder and Baraclow provide the Windows NT login screen as an operating system single point of entry example, since all user access to a Windows NT machine must be through the login screen. Strictly speaking, this is probably not an excellent example, since a number of Windows NT services allow access to the system through other entry points, such as the telnet service and the web service. Perhaps a better example would be the single login system used by Yahoo! and analyzed in [5]. The Microsoft Passport system could be considered an extreme application of the pattern. The UW CSE web site also makes use of the pattern by redirecting access requests for confidential documents to a single authentication point, the "Get Credentials" screen.

The aforementioned web-based applications harnessing the *Single Access Point* pattern all also make use of the *Session* pattern, though one could be used without the other. The *Session* pattern is an extension to the Gang of Four's *State* pattern [6] by adding the notion of a user. Session objects represent per-user global state. This state may include user identification information, user authorization information, and user privileges for security purposes, plus any application-specific user state. The session object acts as a namespace that consolidates globally useful information about the current user.

Web-based shopping carts are a common example of the *Session* pattern in a less obvious security context. Though the user is usually only identified by an anonymous session token,

the confidentiality and integrity of shopping cart sessions must often be maintained in order to ensure customer confidence and protect against interference in the shopping process by active adversaries.

Yoder and Baraclow's *Check Point* pattern for computer security provides a pattern for consolidating security checks and responses into a single place. The authors note that this pattern is very much like an application of the *Strategy* pattern of [6] to authorization logic. Because of this, it allows such logic, which is often based upon volatile security policy documents, to be changed easily. In [8], Mahmoud identifies a very similar pattern called *Security Policy*, and identifies the Java `SecurityManager` class as an example. Because the `SecurityManager` matches the *Check Point* pattern so well, it is a promising interface between application-level and system-level security in Java programs.

The *Secure Access Layer* pattern, like the *Check Point* pattern, is very clearly seen in Java systems. Implementations of *Secure Access Layer* define a single interface through which calls are made to the external environment. This allows external access to be more easily controlled by a *Check Point*, and has the added benefit of insulating the higher levels of the application from changes to operating system and network security primitives and protocols. The Java Virtual Machine provides such a *Secure Access Layer* for Java applications and applets.

The *Role* pattern, described in [1] and [15], is quite useful in managing user privileges in security applications. The idea of roles does not originate with either of these papers, though. As early as version 4.0, Windows NT explicitly used the concept of roles in order to group privileges together and ease user management. Though Unix file system groups are also an example of roles, the explicit nature of Windows NT roles makes them more instructive. In the *Role* pattern, privileges or authorizations are grouped together based upon the role that a user needing those privileges will perform. For example, the Administrator role in Windows NT is a role possessing all privileges. Any user with the Administrator role has full access to the system. Users in the Administrator role may create new roles, with other sets of privileges, and each user is assigned a set of one or more roles. When a graduate student becomes a TA CSE143, that student's user account is given the role CSE143_TA, which grants privileged access to certain file systems and applications on the UW CSE

departmental network. When the student ceases to be a TA, that role is removed from the user account, and all associated privileges are immediately revoked.

In an unpublished article ([4]), Fowler discusses various implementation patterns for dealing with the *Role* design pattern. He encourages application developers to choose the simplest implementation that meets their particular needs, so long as it clearly does meet them. The patterns range in complexity from a type per role to a unified role object representing a relation between two entities (e.g., the role relation worksFor(Megalomart, Andy)).

Romanosky's recent guest feature in Security Focus Online [9] attempts to add several other patterns to those put forth by [15]. His patterns are intended to be useful in the deployment, maintenance and testing phases of an application's life cycle, as well as in the design and development phases. His patterns attempt to address security problems found in dealing with 3rd parties and applications whose security must be considered at many layers in the networking stack. His *Security Provider* pattern is an adaptation of the *Single Access Point* pattern for use by IT professionals, rather than software engineers. It describes a single authentication system shared by multiple web applications, *á la* Microsoft Passport and Yahoo!ś unified login system.

4 Other Security Principles and Guidelines

While not design patterns in the sense of [6], aspects of the thirty-three principles in [12] are patterns for security systems throughout their life cycle. Indeed, some of the patterns suggested in [9] and [15] are design pattern formalizations of these principles. For example, *Risk Assessment and Management*, *White Hats, Trick Thyself* and *Layered Security* appear to synthesize several of the NIST principles. Others, such as "least privilege" and open protocols and algorithms are common design models in computer security (see, e.g., [13], [10]).

The principle of "least privilege" simply states that users should be given no more authorization and no more information than necessary to carry out their tasks. The Unix chroot command, which causes an application to see a different root directory from the actual one, is a way of enforcing least privilege in FTP servers. It works by keeping files that clients

should not be able to access out of the file system namespace of the server. Running services as special dummy users with restricted privileges is another common technique for utilizing least privilege, and it is describably in terms of the *Role* design pattern.

Another very useful design principle in security is, "minimize the system elements to be trusted [12]." Data should never be assumed to be valid, unless the cost of its being invalid is known to be low. In web systems, which are often based upon CGI scripts written in perl or shell scripting languages, this often means that data must be "cleaned" before it may be processed. Perl features an execution mode in which only explicitly cleaned data may be processed. This exists in order to prevent malicious code from being embedded into program input, a common form of exploit in older (and many newer) Perl CGI programs. Servers written in C are notoriously bad at adhering to this design principle. Buffer overruns in FTP daemons surfaced for years because assumptions would be made about the amount of input that a user would give. By supplying more than the assumed maximum, the user could insert arbitrary executable code into the application, or overwrite application files with so-called Trojan horses. Today, though these simple buffer overrun exploits are rarer, they continue to exist, and for that matter, continue to be created anew.

Perhaps the most important principle in IT security is that solutions based upon proprietary protocols and algorithms cannot be considered secure. As Bruce Schneier puts it in *Applied Cryptography*,

> If I take a letter, lock it in a safe, hide the safe somewhere in New York, then tell you to read the letter, that's not security. That's obscurity. On the other hand, if I take a letter and lock it in a safe, and then give you the safe along with the design specifications of that safe and a hundred identical safes with their combinations so that you and the world's best safecrackers can study the locking mechanism—and you still can't open the safe and read the letter—that's security [11].

Systems based upon proprietary algorithms are notoriously insecure. If the strength of the algorithm is its secrecy, rather than the key's secrecy, then once the algorithm is discovered, it has been made useless forever. Such is the problem with the with the technologies designed to protect the intellectual property stored on DVDs. Attempts to circumvent the problems of a poorly chosen, proprietary algorithm via legislation, such as the Digital Millennium Copyright Act, are ineffective, and often out of the public interest [7].

One advantage of using public algorithms is that it reduces the obligation of the system designers to prove the correctness of their algorithm. Though many logics and calculi for proving different protocols and security algorithms correct exist (e.g., [2]), it can be tedious to carry out a proof. Users of published algorithms can rely on publicly available proofs of correctness. Also, when a flaw is discovered in a previously trusted public algorithm, an entire community focuses on correcting it, rather than just a single, ill-staffed design team.

5 Virtual Machines

An interesting area of current research is in the use of Virtual Machines to provide a secure means to efficiently execute untrusted mobile code. Several models for virtual machines exist, including the emulation model of Java, the host system-supported model utilized in VMWare, and the original virtual machine model developed for the IBM OS/360 architecture [pertinent papers]. Each was developed for a different purpose, but all attempt to provide the application with the illusion that it is running natively on its own hardware.

The IBM OS/360 was the first virtualizable computer. The design executed a Virtual Machine Manager (VMM) directly on the hardware. That manager then executed guest operating systems directly on the hardware, trapping privileged instructions for emulation, and periodically pre-empting one guest operating system in order to time-share it with another. If one guest OS crashed, not as the result of hardware failure, the others could continue running. This provided improved reliability on these large mainframes that could not always be achieved at the OS level. It also provided security, since every guest OS operated in its own namespace, and was unable even to refer to the resources of other guests.

VMWare is a modern Virtual Machine, designed to run on x86 architectures. The target use of VMWare is apparently organizations that wish to run multiple operating systems, or multiple versions of the same operating system, on x86 hardware. Its model relies on a host operating system to manage the hardware, saving the VMWare team from the task of writing drivers for all of the plethora of PC hardware devices. It, too, provides security through separation.

The Java Virtual Machine is more emulator than virtual machine, since Java programs do not run directly on native hardware of any real platform. Nevertheless, it provides a sophisticated security model based upon the *Check Point* pattern and a Security Manager. Within a single Java Virtual Machine, several Java applications may execute securely using the same principle of separation used in other virtual machines. Since Java does not have pointer arithmetic, and since the use of multiple class loaders may be used to keep class names separate, there are no namespace conflicts. The only danger, then, is that one application will successfully crash the virtual machine, bringing down the entire system. That this ability to separately run several applications within one virtual machine was designed into Java suggests that specialized Java machines, either real or emulated, might be an effective way to securely execute untrusted mobile code. Already, multiple applications execute within one virtual machine on Java Servlet servers with great success. IBM and several other companies offer commercial virtual machines specialized for this task. A refinement of Java's security management infrastructure to make it easier to understand and use might make this a reality, though the fact that it has not yet happened suggests that sizable challenges remain.

Steve Gribble and his students at the University of Washington are currently developing an x86 virtual machine manager called Denali in order to provide a platform for securely running untrusted mobile computations. The principle of separation is a major motivation for using virtual machines. Another is the relative simplicity of VMMs in comparison to full operating systems. Since the VMM is simple, and sits between the hardware and all guest OSs, the security of the physical system is easier to verify. It is simply necessary to show that access to privileged resources cannot be circumvented [14].

It seems possible that the principle of security through separation might be formally described, and perhaps applied using techniques other than virtual machines and OS processes in order to improve application security. I also posit that a security pattern based upon virtual machines could be developed, perhaps as an extension or combination of the *Check Point* and *Session* patterns. An analysis of how virtual machines relate to other security design patterns should illuminate the particular advantages of virtual machines, and perhaps expose new ones.

6 Conclusions

The explosion of commercial enterprise on the internet in the late 1990s exacerbated the existing need for guidelines, principles and patterns to aid in the development of secure applications. In the application development domain, a number of design patterns have been introduced to aid in the integration of security policies into software during its design and implementation phases. Last year, the NIST introduced a set of principles to be used throughout the system life cycle in order to provide security [12], and since 1999 the Common Criteria for Information Security and others have provided a standard set of evaluation criteria useful for evaluating security in off-the-shelf and custom-built IT applications [3].

Yet, as late as last year, Fu *et al.* published a paper documenting the poor state of security in web applications [5]. Security within some corporations' off-the-shelf mission-critical server software is flimsy, primarily because security concerns have taken a back seat to new features and system performance. Credit card numbers are stolen, and access to network applications is denied. It is easy to be amazed that any information transmitted over the internet is not revealed to all the world.

The problem is an educational one. Information – about individuals, corporations, governments and other organizations – is worth more today than at any time in history, and the cost of acquiring it via espionage has dropped precipitously thanks to the ubiquity of the internet. Most individuals, though, are not equipped to understand the value of their Social Security number, or the weakness of protecting their financial profile with their mother's maiden name or their favorite color.

Even once people understand the value of the information that they send daily across the internet, those who develop applications will still need to learn enough principles of basic information theory and computer security to provide systems that protect it. The authors of [12] leave as their ultimate principle that application developers must be trained properly. Pulling no punches, they state that, "it is unwise to assume that developers know how to develop secure software (p. 19)."

The conclusion from this is that the most important direction for research in computer security should be toward better pedagogical techniques and easier-to-use and easier-to-

understand security techniques. Many computer systems running operating systems with highly sophisticated security systems are never properly secured because the task of learning the security measures is so daunting that few ever do, and fewer ever do it correctly. Many techniques that could enhance our ability to keep personal data confidential and our communications tamper-proof are not deployed because people do not understand their benefit well enough to demand it. Until that changes, all other enhancements to data security are superficial, at best.

References

[1] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The role object pattern. In *Proceedings of the 4th Pattern Languages of Programs Conference*, 1997.

[2] Michael Burrows and Martín Abadi. A logic of authentication. In *ACM Transactions on Computer Science*, volume 8, pages 18–36, 1990.

[3] Common Criteria. http://www.commoncriteria.org.

[4] Martin Fowler. Dealing with roles. Working Draft, see http://citeseer.nj.nec.com/fowler97dealing.html, July 1997.

[5] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. Dos and don'ts of client authentication on the web. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[7] Andrew Grosso. Legally speaking: Why the Digital Millenium Copyright Act is a failure of reason. *Communications of the ACM*, 2002.

[8] Qusay H. Mahmoud. Security policy: A design pattern for mobile Java code. In *Proceedings of the 7th Pattern Languages of Programs Conference*, 2000.

[9] Sasha Romanosky. Security design patterns. *Security Focus Online*, January 2002.

[10] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 1974.

[11] Bruce Schneier. *Applied Cryptography: Protocls, Algorithms and Source in C*. John Wiley and Sons, Inc., New York, New York, second edition, 1996.

[12] Gary Stoneburner, Clark Hayden, and Alexis Feringa. Engineering principles for information technology (a baseline for achieving security). Technical Report SP-800-27, National Institutue of Standards and Technology, Gaithersburg, MD 20899-8930, June 2001.

[13] D. Wheeler. Secure programming for Linux and Unix HOWTO. http://www.dwheeler.com/secure-programs/, 2000.

[14] Andrew Whitaker, Marianne Shaw, and Steve Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, University of Washington Department of Computer Science and Engineering, 2002.

[15] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Proceedings of the 4th Conference on Patterns Language of Programming*, 1998.