Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials: Network, Endpoint, and Cloud (Security 401)"
at http://www.giac.org/registration/gsec

Designing Security into Software with Patterns

Alfred W. Amos
GSEC Practical - version 1.4b
April 26, 2003

Abstract

Software is a major problem when trying to provide a secure computing environment.
Vulnerabilities from software defects can compromise a company's resources and
damage their reputation through loss of customer records. The best solution is to
eliminate all vulnerabilities through good coding practices and software testing. In
addition, legacy software needs to be re-engineered to protect existing code
vulnerabilities from exploits. In today's programming environment, this is unrealistic due
to time, budget and knowledge constraints.

Many Information Assurance tools have been designed at the host and network level to
mitigate this problem. They work well to address specific security concerns, but fall
short of eliminating software exploits. More direct methods need to be applied by
building security measures into the software. The concept of "Best Practices" can be
applied with patterns to help develop secure software. Identifying security patterns is an
area of active research to capture common solutions for security issues. Design
patterns have been used for years to describe common designs for source code
implementation. In order to make software secure, it is also necessary to use
implementation guidelines with adequate testing procedures. Patterns with
implementation guidelines can provide a powerful tool for developing software with built
in security.

Software with Embedded Security

In an article by McGraw on building secure software, the author states "software is the
biggest problem in computer security today"[1]. The complexity of software systems,
operating in hostile environments like the Internet, often lead to attacks exploiting known
or suspected vulnerabilities. A well-known example in recent years is the Code Red II
worm[2]. This worm exploited buffer overflow vulnerability in the Microsoft IIS Server,
opening up a backdoor allowing a remote intruder to run arbitrary code. Many factors
contribute to the problem of insecure software. Client processes using the system do
not always behave as expected and access the services in unpredictable ways.
Software developers may lack necessary expertise contributing to poor software design
and unsafe code implementation. Program managers may not understand the issues to
justify sufficient resources, time and budget. Tests performed on the system may be
incomplete, especially when new features are added or the operational environment
changes.

---

[1] McGraw.
[2] CERT.

Embedding security into software may add complexity, but several benefits may justify the effort. The risk associated with software defects may be reduced to a level that management can tolerate. Management needs to understand how the implemented measures will provide protection so resources, time and budget can be allocated for development. If management is not convinced, the security may be minimized or excluded. Software components supporting security features could provide protection when other measures fail. If data logging is included as a security measure, the application can be monitored during operation. Information Assurance (IA) enabled devices, within the infrastructure, might use this data to give better diagnosis of related events. Embedding security into software and sharing information with tools in the trusted computing base can contribute to a more secure environment.

<u>Reliability through "Best Practices"</u>

Successful designs for security problems and software implementation have been used for many years. Reusing and adapting these designs to new systems can make development easier, more efficient and more reliable. Patterns have become popular as a way to capture and present "common solutions to a recurring problem in a structured format"[3]. Patterns represent best practices, since they are discovered from proven solutions in the industry.

Security patterns describe design and procedural techniques for solutions to specific security concerns. They can help educate developers by providing a template for successful design. Developers, without knowledge of security, must work closely with security engineers to build secure software. With the availability of patterns, security engineers may only need to oversee the development and provide support for special problems. Many examples of security patterns can be found in the Security Patterns Repository[4] and at other sites on the Internet.

Design patterns describe techniques that have been successfully applied for solving software problems that occur repeatedly. The same or similar software techniques have been reused and often reinvented for different software development efforts. Capturing and describing these patterns in a structured format makes them more readily available for reuse. A pattern describes a general design problem and not a specific implementation. An architecture design can be constructed with patterns and refined through iterations of the development cycle.

Anti-patterns are closely related to design patterns but yield the opposite effect. Developers may always have good intentions, but choosing the wrong pattern or combination of patterns may lead to software with bloated interfaces that does not work as expected. A complex system with too many levels of indirection may result, making the system slow and unusable. Keeping the design simple and adding the minimum

---

[3] Kienzle, "Security Patterns Overview."

[4] Kienzle, "Security Patterns Repository."

number of components necessary for needed functionality helps to keep the code size small and verifiable.

Security Patterns

Three security patterns extracted from the Security Patterns Repository[5] have been adapted to describe user authentication, data filtering and log auditing. The following format is used for the discussion.

Context – States the purpose of the security pattern.
Problem – Describes the problem the pattern is attempting to solve.
Solution – Specifies how the pattern can be used to solve the problem.

Authenticated Session[6] security pattern provides secure access control from untrusted clients with a single logon.

Context: The Hypertext Transfer Protocol (HTTP) is considered stateless because each transaction is independent. This protocol can be used to provide Extensible Markup Language (XML) documents and Web pages for e-commerce and other applications. In many cases a site may want to have access restrictions to protect the data stored on the servers. Requiring separate authentication for each transaction can make the service unusable from the users perspective. The authenticated session pattern provides access to the service with a single authentication for a single session and allows multiple transactions in a stateful manner.

Problem: If the security policy allows a user to authenticate once per session, the user's identity and session information must be maintained between requests. One way to do this is by passing the information between the server and client for each transaction. The session information stored on the client system in the XML document, Web pages or cookies may create considerable risk depending on how the client treats it.

Solution: The amount of authentication information transferred between the server and client over the Internet should be minimized. Storing the user's identity and session information on the server provides better assurance that data tampering has not occurred and helps protect the user's privacy. The client only needs to have minimal information, such as a session identifier that can be generated randomly on the server. When the user makes a new transaction on a previously established session, the identifier is sent with the request. Since the session identifier is the key to identify the user and access information on the server, it should be kept safe during the entire session. A protected connection, using Secure Socket Layer (SSL) or Transport Layer Security (TLS), is suitable for this purpose. Encrypted storage on the client system may also be required. When the session is finished the server invalidates the session identifier requiring

---

[5] Kienzle, "Security Patterns Repository."
[6] Kienzle, "Security Patterns Repository, Authenticated Session."

the user to re-authenticate. Session data held on the server could include start time and last access time. These times are needed to install a timeout mechanism to terminate and clean up the session when the connection is abnormally closed. A detected trend relating to abnormal terminations could raise an alert and trigger an appropriate response consistent with policy.

Client Input Filter[7] security pattern can be used for data filtering after user authentication has been performed.

Context: This pattern protects a server against invalid data from a user. Incoming data from a client may be constructed to circumvent security on the service, exploiting suspected vulnerabilities. Invalid data with no malicious intent might also be sent to the server.

Problem: If security measures are enforced on the client side, inspection of the code could betray information on what protective measures have been deployed. A client could then launch an attack bypassing the client side security. This pattern assumes all data from a client is suspect and should be filtered at the server.

Solution: A filter is used to check for invalid input data on the server side before it is sent to the service. If checks are performed on the client system, they should be rechecked on the server. Sensitive data saved on the client should be stored in an encrypted format. The Client Data Storage[8] pattern suggests symmetric encryption, as a less expensive solution, but if asymmetric negotiation only needs to be done once, the cost of public key encryption might be tolerable. Incoming data can be modified on the server before continuing to the service. Questionable data could be logged and rejected. These logs may reveal patterns of attempts to circumvent the security of the system. In order to help identify suspicious client behavior, the server could check the input data for content like unexpected data field values, random garbage and other similar requests. A variation of this pattern could also check the output data from the service to ensure protected information is not sent back to the client.

Log for Audit[9] security pattern is useful for recording and auditing information from events occurring during the service operation.

Context: Software applications usually run as expected when input is valid and received in the correct format. If this is not the case, a program may crash and produce incorrect results or perform unauthorized actions. It would be beneficial to know what data was responsible, as well as when and where it happened. In the case of a malfunction or an exploit, logging information provides an audit trail

---

[7] Kienzle, "Security Patterns Repository, Client Input Filter."
[8] Kienzle, "Security Patterns Repository, Client Data Storage."
[9] Kienzle, "Security Patterns Repository, Log for Audit."

for investigation. Recording events during operation also help to demonstrate that a service is secure and available.

Problem: Multiple log repositories can be difficult to manage, access and protect. The amount of data captured, type of events logged and the software code components responsible for logging are important considerations. Recording too much data may cause the log files to overflow and critical events may be lost or overlooked. On the other hand, if too little data is recorded, information needed to determine a problem may not be available. In either case, it is difficult for auditing to be effective.

Solution: Data should be recorded in a central location for simplicity, accessibility and protection. Events and information must be logged from key components in the code. Logs can give assurance of correct operation by providing user accountability, reliability monitoring and performance measurements for security and non-security related purposes. Security related logs should be routed to a separate repository for controlled and guarded accessibility. Good auditing practices need to be implemented to make logging effective. An analyst should examine the logs regularly for proper use and operation of the service. For automatic auditing, the software might check for multiple logins from the same user and password attacks through excessive attempts. Invalid input from the client or inappropriate output from the service could be recorded and trigger an auditor to terminate the session or perform other actions consistent with policy. The type of event and the code section logging the event is useful information on an attack. Information collected at the application layer might also be correlated with data logs from IA enabled devices for further clarity on suspected events. In addition, results from the auditor could be used to cue lower layer protection mechanisms for diversionary or exclusionary actions.

<u>Design Patterns</u>

The following design patterns have been chosen to help implement the security issues discussed above and to define an initial architecture. These are only brief descriptions of the patterns. The full descriptions can be found at the specified source.

Façade[10] pattern provides a simpler single interface to a complex system with multiple subsystems. It defines an entry point and coordinates the access to these subsystems.

Wrapper Façade[11] pattern is similar to the façade pattern and may consist of several classes. This pattern works with lower level operating system functions and applications that are not object-oriented. An example might be to encapsulate the lower level functions for socket connections, providing an object-oriented interface with abstraction and data-hiding benefits.

---

[10] Gamma.

[11] Schmidt, "Wrapper Façade, A Structural Pattern for Encapsulating Functions within Classes."

Adapter[12] pattern defines an interface that a user expects to access another class. This interface may be different than the target class provides. If two or more classes have different access methods, an adapter can provide each of these classes with common access capability.

Proxy[13] pattern serves as a surrogate for another class to control access. This is similar to an adapter pattern, except its access methods will be the same as in the target class. A common use of this pattern is to do additional processing or filtering of the data for input before passing it to the service and for output before the response is sent to the client.

Singleton[14] pattern provides a single point of entry to a target class. There should be only one instance of a Singleton object. The address of the object is stored in static memory and is easily available to other objects through a static function call. Singletons are not thread safe by default. Upon initial creation of the object a race condition could occur. Multiple threads trying to access the Singleton simultaneously will result in multiple objects of the class being created. Thread safety can be achieved, in this case, through use of the Double-Checked Locking[15] pattern. Access to data in this class should also be protected from race conditions. Since this pattern provides a single point of access from multiple threads, efficient processing is necessary to avoid making it a chokepoint.

Double-Checked Locking[16] pattern describes an efficient way to ensure that only one Singleton object can be created.

Orphaned Thread Bug[17] pattern provides a way to notify dependent threads if the master thread were to become unavailable. The termination of a master thread may cause a program to halt, if dependent threads exist. Other effects could include a possible exploit opportunity and unavailable computing resources due to a self-imposed denial of service attack, from a runaway thread. With exception handling, dependent threads can be notified of the condition and respond by terminating after logging the event.

Defining the Architecture

The architecture in figure 1 shows a possible implementation of the three security patterns described above. Five main functions represented in this architecture are user authentication, data filtering, event logging, log auditing and the service. The following discussion describes each of the components in the diagram.

---

[12] Gamma.
[13] Gamma.
[14] Gamma.
[15] Schmidt, "Double-Checked Locking."
[16] Schmidt, "Double-Checked Locking."
[17] Allen.

The Service component provides the main function of receiving the user's request and generating the response, after the other components authenticate and filter the transaction. The type of service intended is non-security relevant, such as accessing XML documents from a database or generating Web pages. For security relevant services, additional measures expressed may undermine the intended functionality. The system would become more complex with larger code size, possible introduction of vulnerabilities and more difficult software verification. This could lower the trust level for the security mechanisms in the service.
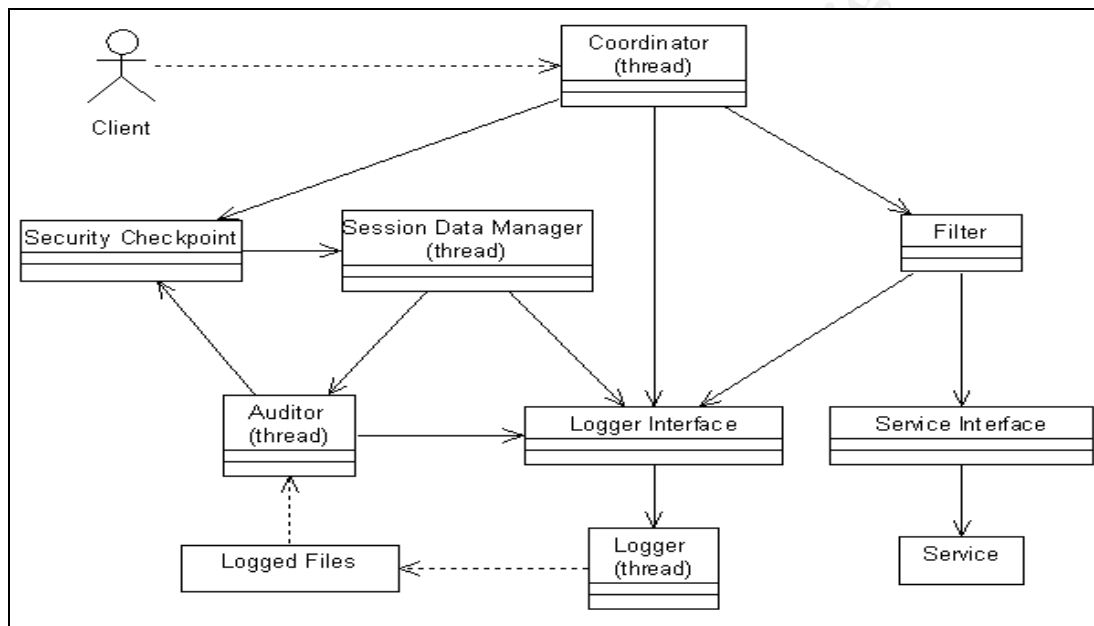

Figure 1

The Coordinator component is the entry point into this system. A request from the client is first routed to the Security Checkpoint and then forwarded to the Filter only if access were granted. If access was refused or the Filter returns an invalid response, the event could be logged and the transaction terminated. A Façade pattern would be helpful here to coordinate and dispatch the messages to other components.

User authentication can be represented by the Authenticated Session[18] pattern. The main components are the Security Checkpoint and Session Data Manager.

>   The Security Checkpoint might be a Singleton pattern that provides a single point of entry to access the Session Data Manager. This objects main objective is to either create a Session Data Manager for a new user or find the correct manager for the given session id.

---

[18] Kienzle, "Security Patterns Repository, Authenticated Session."

The Session Data Manager component may consist of multiple classes and handles the authentication for a single session. It would store all the identification and session data for the user. The session id based on the client's password can be generated in this component. In a more user-restricted environment, additional authentication might be used. Two-factor authentication might be implemented requiring the client to provide a security token or client certificate in addition to a password. A timeout mechanism can close the session, if the connection terminates abnormally. When the session is terminated all cached information about the session and user should be flushed. All or part of this information could be logged, as desired.

Data filtering can be implemented using the Client Input Filter[19] pattern. Key elements include the Filter and Service Interface components.

The Filter might be represented by a Proxy pattern defining a surrogate for the Service Interface component. In the absence of the Service Interface, Filter would be a proxy for the service. One reason for placing the Filter proxy before the service is to add a layer of protection by checking for invalid requests from the client, as well as for inappropriate responses from the service. A malformed crafted request could be stopped before reaching the service. If a response from the service includes restricted data, the Filter could stop the transaction. In either case, the Logger can record the events. This releases the service from performing the filtering operation.

Service Interface is optional depending on the type of service being installed. This component might take the form of a Façade, Wrapper Façade or an Adapter.

Event logging and log auditing is addressed by the Log for Audit[20] pattern using the Logger Interface, Logger and Auditor components.

Logger Interface might be a Singleton pattern providing a single point of entry to the correct Logger object.

The Logger is intended to record events from the other components into a centrally located Logged Files repository. For instance, all successful and unsuccessful logins could be recorded from the Session Data Manager. A well-designed logging system can provide good metrics for analysis. The event logger component would consist of one or more classes to perform the necessary functions. A separate Logger can be created for each client, using the session id.

A separate Auditor could be created for each Session Data Manager. The purpose of the auditor is to monitor the recorded data only for a single session. If access was refused multiple times during authentication the Auditor could notify

---

[19] Kienzle, "Security Patterns Repository, Client Input Filter."
[20] Kienzle, "Security Patterns Repository, Log for Audit."

the Session Data Manager to lock out the user. For information on lockouts, see the Account Lockout[21] security pattern at the Security Patterns Repository. This could help prevent a denial-of-service attack by flooding the system with bad authentication credentials. The Auditor component could also be responsible for notifying an operator and other security relevant devices for further analysis and action.

The Logged Files component represents a single repository for storing information recorded by the Logger objects.

As this architecture has been described, it allows only one user at a time. Introducing threads can improve availability by accommodating multiple users. The Coordinator might run on its own thread for each request from the client. The thread would terminate after the response is sent back to the client. The Session Data Manager could also run on its own thread, so it will not terminate with the Coordinator thread. Using separate threads for the Auditor and Logger will allow them to operate independently from the main path and prevent excessive delays. However, the use of threads introduces additional problems. This may require other patterns and structures to be implemented to monitor and control the threads. For instance, the Orphaned Thread Bug[22] pattern provides a way that a terminating thread can notify dependent threads, so appropriate action can be performed.

From Design to Implementation and Test

Security patterns provide a solution for a security problem, while design patterns describe the components to implement the solution. Additional issues that need to be addressed include source code development, test, verification and validation. If the code is written without security in mind, applying best practices for design will be of little help in building secure software. Implementation guidelines enforced through policies can assist programmers in writing the code. Some of the guidelines that have been commonly used for years are discussed below.

Using the right tools is expressed in the Choose the Right Stuff[23] security pattern. It describes how development can be made easier with a better chance of success if the right language, libraries and tools are used for the development. For example, Java may be better than C or C++ for security but may not be as efficient. Picking the right language and tools may be difficult due to the lack of language expertise, available compilers, required legacy software and the target hardware to run the application.

Good programming practice is essential to avoid vulnerabilities that could lead to exploits. Dangerous functions in languages such as C should either be avoided or carefully checked for suitable implementations. Bounds checking and ensuring

---

[21] Kienzle, "Security Patterns Repository, Account Lockout."
[22] Allen.
[23] Kienzle, "Security Patterns Repository, Choose the Right Stuff."

that buffer overflows do not occur help considerably in making software secure. Manual and tool based source code auditing could help with this effort. A major goal in good programming practices and creating secure software is to keep the design and implementation simple. This helps in understanding the code and makes the task of software verification and validation easier.

Exceptions can help avoid a software crash and intrusion. Unexpected events can cause exceptions to be thrown and route the flow to alternate processing. Exceptions should be used for exceptional circumstances and not for normal code logic, so runtime efficiency is not impacted. This could help ensure availability of the service and provide information on what is happening through data logging. Appropriate action in response to an exception could be performed manually by an operator or automatically by the code.

Threads help ensure availability and efficiency, but if implemented incorrectly, they can create vulnerabilities. Constructs and patterns need to be implemented in a thread-safe manner. Monitors can be constructed within the code to help dictate thread operation. Constructs accessed by multiple threads need to be synchronized allowing one thread at a time, avoiding race conditions. This adds complexity, but it may be justified by the benefits.

As with all software, testing is a crucial step in ensuring it works as expected and provides a level of assurance for secure implementation. Patterns provide detailed documentation of the architecture design and implementation. By analyzing the architecture, critical sections of the system that present a significant risk might be identified. Certain components provide a specific role and can be tested on how they perform that role. These areas could receive additional manual code inspection beyond that used for other components. Threats might also be simulated against these areas for more comprehensive testing. In the system above the areas of most concern might be the Security Checkpoint, Session Data Manager and Filter to protect the authentication and filtering mechanisms. In addition, the Coordinator should be checked for the possibility of misrouting a request and the Logger Interface must keep the Logger safe from attacks. In order to establish a high assurance that the software is secure, verification for compliance with the security requirements and validation of operation at the acceptable level of risk is necessary.

These measures can help considerably to keep software available for use, but sometimes the inevitable happens. If the application does crash, it should fail securely and terminate the sessions and cue connections. Making the software run with the least privilege and least duration of privilege can help reduce the risk of a compromise. After a crash, services should be resumed only after authorized diagnosis and remediation of the problem. This may require some remedies at Layer 2 and 3 for proper enforcement.

At some point there may need to be a trade between code safety and performance. When is the software secure enough? This can be a difficult question to answer and

requires a risk analysis against the perceived threats. Through operational auditing of logged information, metrics can be analyzed to decide if security should be added.

Conclusion

Developers have been using patterns for years, often without realizing it. Since patterns are discovered from proven solutions, they represent best practices in the industry. Descriptions of these practices in a structured format enable their use for training, for documentation and as a means for communication. Patterns promote the concept of reusable components, which could shorten the time to a finished product with decreased cost. They help create an architecture that is harmonious with the business risk model that can be adjusted and refined during the software development cycle to achieve the fidelity and control necessary and specified. Patterns enable a tester to focus on critical portions of code, enhancing threat determination and risk evaluation. Experience in using patterns helps considerably to avoid bloated code, inefficiency and complexity. Keeping the components small, simple and verifiable contribute to high assurance for a trusted system. Software with built in security can complement traditional network and host based security measures by adding layers as in "Defense in Depth".

As mentioned earlier, the code implementation is critical. Without secure implementation a secure design is of little value. Guidelines for implementing safe code can also be captured from past experience. They can help eliminate vulnerabilities and create bug-free code. In order to ensure secure implementation, adequate testing needs to be done at several levels from source code audits to tests in the operational environment. Source code audits should be done through manual inspection and automated tools. Verification and validation, of the security implementation, is necessary to establish a high assurance for correctness and completeness. If all software were written with secure design and good implementation guidelines, software would not be the focus of concern it is today.

References

Allen, Eric E. "Diagnosing Java Code: The Orphaned Thread bug pattern."
August 2001.
http://www-106.ibm.com/developerworks/java/library/j-diag0830.html.

Barkley, John. "Principle of Least Privilege." January 9, 1995.
http://hissa.nist.gov/rbac/paper/node5.html.

Blakley, Bob. "Security Design Patterns (SDP)." Open Group's Security Forum.
January 28, 2003. http://www.opengroup.org/security/gsp.htm.

Brown, William, et al. "What's an AntiPattern." AntiPatterns. January 24, 2000.
http://www.antipatterns.com/thebook.htm.

CERT. "CERT Advisory CA-2001-13 Buffer Overflow In IIS Indexing Service DLL." January 17, 2002. http://www.cert.org/advisories/CA-2001-13.html.

Department of Defense. "Roadmap for Developing an MDA-Facilitated Information Assurance Architecture", Version 1.2. April 22, 2002. http://www.omg.org/docs/c4i/02-04-03.doc.

Gamma, Erich, et al. Design Patterns, Elements of Reusable Object-Oriented Software. Reading MA: Addison-Wesley, 1995.

Hays, Viviane, Marc Loutrel and Eduardo Fernandez." The Object Filter and Access Control Framework." 2000. http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/Fernandez3/Fernandez3.pdf.

Heaney, Jody, et al. "Information Assurance for Enterprise Engineering." 2002. http://jerry.cs.uiuc.edu/~plop/plop2002/final/PLoP-2002-Heaney-7-22.pdf.

Hooker, David. "Keep It Simple." 2003. http://c2.com/cgi/wiki?KeepItSimple.

Kienzle, Darrell, et al. "Security Patterns Overview." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/.

Kienzle, Darrell, et al. "Security Patterns Repository." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/repository.html.

Kienzle, Darrell, et al. "Security Patterns Repository, Account Lockout." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/getPattern.html?name=Account%20Lockout.

Kienzle, Darrell, et al. "Security Patterns Repository, Authenticated Session." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/getPattern.html?pid=2.

Kienzle, Darrell, et al. "Security Patterns Repository, Choose the Right Stuff." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/getPattern.html?name=Choose%20the%20Right%20 Stuff.

Kienzle, Darrell, et al. "Security Patterns Repository, Client Data Storage." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/getPattern.html?name=Client%20Data%20Storage.

Kienzle, Darrell, et al. "Security Patterns Repository, Client Input Filters." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/getPattern.html?name=Client%20Input%20Filters.

Kienzle, Darrell, et al. "Security Patterns Repository, Log for Audit." Networks Associates Technology, Inc. 2002. http://patterns.nailabs.com/getPattern.html?name=Log%20for%20Audit.

McGraw, Gary. "Building Secure Software: Better than Protecting Bad Software." January 2003. http://www.sqmmagazine.com/issues/2003-01/bss.html.

Schmidt, Douglas C, "Wrapper Façade, A Structural Pattern for Encapsulating Functions within Classes." February 1999. http://www.cs.wustl.edu/~schmidt/PDF/wrapper-facade.pdf.

Schmidt, Douglas C, and Tim Harrison. "Double-Checked Locking." 1997. http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf.

Schmidt, Douglas, et al. Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects, Volume 2. New York: John Wiley & Sons, April 2001.

Viega, J. and McGraw, G. Building Secure Software. Boston: Addison-Wesley, 2001.