Global Information Assurance Certification Paper

M. Stuart Perkins
February 01, 2004
GSEC Practical Assignment, v. 1.0 (Initial Release)

Design From a Distance:

An Introduction to Security Design Patterns

TABLE OF CONTENTS

Abstract

Recent studies have shown that human error is the number one cause of security breaches resulting in real or serious harm to major corporate networks. Companies do not have the time or resources to train their entire IT staff to the level of technical competency of a certified security professional, and yet, the average IT professional is responsible for the security of the networks on which they operate and the software that they write. What is needed is a way to capture the knowledge and experience of professionals and make it available to everyone in a language that does not obfuscate principles with industry lingo and technical detail.

Design patterns provide just such a language. They describe generic approaches to solving recurrent problems that are not tied to any particular language or implementation. The use of design patterns can help a software developer or security professional avoid costly mistakes, save time on design, and think through a problem on an abstract level.

This paper introduces the Design Patterns methodology, and presents several basic security patterns, including a pattern that acts as a foundation for the design of any security system, the Design From A Distance pattern.

Problem Statement

In 2003, the Computing Technology Industry Association (CompTIA) commissioned a study on the state of network security within major corporations. Matthew Poyiadgi, a regional director for the association found the results "pretty staggering" [Sturg03]. According to an article by Will Sturgeon on ZDNet, March 2003, "Human Error in the workplace still poses the single largest threat to corporate networks, with a lack of training being blamed for a problem which businesses should have overcome long ago."

The findings of the study indicate that technology alone is not sufficient for securing a network against today's sophisticated attacks. It is becoming more important than ever for companies to maintain a security-conscious staff. More than two thirds of the companies polled claimed that only a quarter of their staff is trained on security issues [Sturg03]. Obviously, not everyone can become a certified security expert, certainly not on the company's time (or the company's dime), but every IT professional must understand fundamental security principles, because as demand for security

experts outgrows the supply, management will depend more on the rest of their IT staff to keep their networks safe. Imagine this scenario:

> You are a software developer. You've worked for your company for quite a few years and have gained the respect of senior management as one who is trustworthy, competent, and able to rise to any challenge. Due to budget cutbacks, and disappointing revenue projections, your company must scale back its payroll. They lay off the entire systems administration staff at your office and ask you to take over the responsibility of network security as a *collateral* duty.

It may sound unlikely, but it happened to a friend of mine just recently. How does one who has no background in network security begin to plan a security policy for a large, distributed corporate network? Fortunately, when you strip a good security policy of industry-specific semantics, and remove all references to specific tools or implementations, what remain are axioms and principles that would make good sense to anyone. Therefore, the first rule of thumb to planning a network security policy, for the non-expert, is to separate the design from the details, or in other words, design from a distance.

Introduction to Design Patterns

Design from a Distance (The Value of Patterns)

There has been a lot of hype about Design Patterns in recent years, especially among software engineers. Even more recently, organizations like the Open Group(www.opengroup.org) have begun applying the Design Patterns methodology to security systems [Open Group]. Strangely, even some of the loudest advocates of Patterns would have a hard time describing what they are and why they're useful.

So why are they useful, anyway? Imagine you're taking a hot-air balloon ride. When you first climb into the bucket, you look down. You see blades of grass, the occasional wildflower tossed in. If you were an avid gardener, perhaps you could even pick out the different types of grass there. As the balloon begins to rise the blades of grass become more numerous as your field of vision grows larger. Eventually, you can no longer think of the individual blades, only that together they make a field. Other objects come into view. Trees become forests. Houses become towns. As your mind receives more input, it finds ways to group objects together so that it can

catalog them more easily. As you distance yourself from a problem you can see the big picture, and think about what the forces, components, and interactions are which define the problem on an abstract level.

This is the primary advantage of patterns, and it explains why they were such a success when introduced to the software design world in the early 1990s. By that time, software had become extremely complex and development lifecycles were getting longer. Developers were spending too much time in the details, and often these details were the same solutions to the same problems that had already been faced and solved a million times. Patterns made it possible for common problem/solution pairs to be identified, labeled, and categorized, completely independent of language or implementation, so that developers could think through the design without worrying about the details.

After the Internet explosion in the late 1990s, the complexity of networks, and therefore network security, began to grow at an exponential rate, so, naturally, the security community has begun to adopt a pattern-based approach to securing large, complicated networks.

A Definition of Design Patterns

It is important to note, before continuing any further, that the key word in any discussion of Design Patterns is *abstraction*. Patterns are not meant to be solutions, rather, approaches to solutions. They are independent of any language or implementation. Consider the following definition by Christopher Alexander, author of *The Timeless Way of Building*.

> Each pattern describes a problem which occurs over and over again in our environment, then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. [Alexander79]

If it seems vague, that is because it was meant to. The first step to understanding Patterns is to separate oneself from the details. (Remember, design from a distance.) Alexander was speaking in the context of buildings and bridges but the same applies to network security [Hohmann03]. The knowledge and experience that is essential to the career of a security professional has nothing to do with syntax or semantics, but the larger conceptual constructs that are implemented using that syntax[GoF94]. For

example, which is more valuable to a DBA, knowledge of MySQL or knowledge of relational database concepts?

A Pattern Language

There are four essential elements to a pattern [OG02]:

1. name – This is arguably the most important part, even though it's the shortest. The name should effectively describe the usefulness of the pattern in a few words. It should be memorable so that it can be easily called to mind.
2. problem – This may be one problem or a system of problems that the pattern addresses. It can be complicated, but the idea is to describe, in generic terms, the rationale behind the pattern.
3. solution – The solution portion of the pattern should be "specific but flexible"[OG02]. It is important that this solution doesn't bind to any language, tool, or procedure. It should just describe the elements and interactions among the elements of a typical solution.
4. consequences – This section should list the trade-offs among competing forces when using this pattern.

These four basic elements represent the core of a good pattern; however, most pattern languages offer a much more restrictive format or template [Kienzle02]. The elements of a pattern may be represented using graphs, use cases, UML diagrams, or text. Often, the problem and solution sections are broken up into several subsections such as *intent, motivation,* and *applicability*. A well-written pattern is thorough, addressing all relevant aspects of the problem domain, and conveys practical experience [GoF94]. There should be examples cited of how this pattern has been useful in the past. In the case of software design patterns, there may even be sample code as an example of how the pattern might be implemented for a specific language.

There are many variations on the pattern template set forth by the Gang of Four (GoF) in their book, *Design Patterns*. The one used in this paper is based on that one, but with a few elements removed for simplicity's sake, and the one element added. The template looks like this:

Name: the pattern name
Alias: the set of alternate names for this pattern
Intent: a short description of the purpose of the pattern

Motivation: a description of the problem domain, with the intent of showing the need for this pattern. Often this will explain how this pattern addresses a fundamental security principle.
Principles: there are some security axioms that have been floating around since the mid 1970s (possibly longer) that form a foundation for good design. The principles addressed by this pattern should be listed here.
Applicability: description of the cases in which this pattern is useful
Forces: the set of forces that describe the way the system should work or behave and that may effect the implementation
Solution: a description of how to approach the problem. This should include implementation caveats, practical examples, and diagrams if applicable. This section tends to be lengthy.
Consequences: the trade-offs among competing forces
Related Patterns: other patterns that may be used in conjunction with this pattern or in its place

The Enclave Pattern

The following security pattern forms a foundation for thinking about securing a network. It is a simple reminder of what a secure or protected system really is, on an abstract level. Notice the use of non-technical language to describe the forces involved. Essentially, there are resources that must be protected, a resource manager that has access to these resources, a protected place or enclave in which these resources reside, an access policy which dictates who can and who cannot access the protected resources, and a guard who is responsible for enforcing that policy [OG02].

Name:
Enclave
Alias:
Reference Monitor
Protected System
Intent:
Structure a system so that all access by clients to resources is mediated by a guard which enforces a security policy.
Motivation:
Some systems contain resources that must be protected. The owner of the system depends on the confidentiality, integrity, and availability of the resources on that system. If any one of these aspects for any one resource is

compromised, the system itself is considered to be compromised, and all resources within it, suspect.

Principles:

Compartmentalize

Applicability:

Use this pattern whenever you want to control access to a system through a single access point using a guard mechanism to enforce an access policy.

Forces:

- Client or Requestor
  - Submits entry request to the Guard
- Guard
  - Mediates all requests for entry using an Access Policy
  - Grants access to Clients that are accepted by the Access Policy
  - Denies access to Clients that are not accepted by the Access Policy
  - Cannot by bypassed
- Resource Manager
  - Services requests to access protected resources for Clients that have been allowed in by the Guard
- Access Policy
  - A rule set by which requests from Clients are validated
  - The access policy must be allowed to change without effecting other system forces
- Access Point
  - A single point at which the Guard protects the system
  - The system has one and only one Access Point

Solution:

The trick is ensuring that the system's Guard is non-bypassable. A firewall, for example, may technically be bypassable if users within the network can hook up a modem and access the internet through their ISP. In order to make the firewall non-bypassable, the network administrator can institute and enforce a strict policy against modems behind the firewall using tools like Phone Sweep (available at www.sandstorm.net). Other methods of bypassing firewalls include peer-to-peer file sharing with tools like Gnutella, which takes advantage of the fact that a typical firewall configuration will not prevent a node inside the network from initiating a connection outside.

In the case of a software application, the Guard may receive some input like a password directly from the user. In this case, it is imperative that

the Guard is not "corruptible" in the sense that it could be bypassed by malicious data passed in by the user. A data-poisoning attack may be used to take advantage of a Guard's inability to deal with invalid inputs. Such data may result in a buffer overflow, allowing malicious code to be run within the protected system. Such vulnerabilities can often be avoided by vigilant programming.

Consequences:

Resources are isolated

Loose coupling among forces [OG02].

- Notice that if the Guard mechanism or Access Policy changes, no other forces are affected

Degrades Performance

- It's a fact of life. Whenever you put an authentication mechanism between clients and data, the request cycle slows down, sometimes dramatically.

Related Patterns:

GoF defines this pattern as a "Protection Proxy" because it describes the mediation of requests for services by a Guard [GoF94].

Security Patterns

Overview Of Security Patterns

Security patterns provide the following advantages according to Schumacher [Schumacher01].

1. Beginners gain the power of expert experience
2. Problems and solutions can be identified and paired more efficiently
3. Problems are approached in a structured way
4. Dependencies of forces can be discovered and handled appropriately.

The example given above is a simple one but illustrates the power of approaching a problem in a structured way. The discussion of a protected network as a system of forces brings to light an important design element: The Guard is separate from the Access Policy. This allows the Guard to be swapped out easily if it is determined that it is bypassable or if it fails in some way. Consider modern Windows security. The Kerberos protocol acts as the single entry point, or Guard, to the domain controller, but may be

swapped out and replaced by NTLM for backward compatibility without affecting the rest of the system.

Enclave is an example of an *entity* pattern, one of three types of security patterns that work together to provide a useful *system of patterns*. Other types of patterns include *structural* and *procedural* patterns. They are defined below along with some examples. They all address one or more of the security principles first outlined by Saltzer and Schroeder in 1975 [Saltzer75], a few of which are listed here.

1. Secure the weakest link
2. Practice Defense in Depth
3. Fail Securely
4. Follow the Principle of Least Privilege
5. Compartmentalize
6. Keep it Simple

Structural patterns are definitions of key system design concepts that describe correct usage of pre-defined entity patterns. They tend to be hierarchical, in that more complex structures can be broken down into simpler structures. Structural patterns are formulated such that their implementations are actual "products" that can be deployed. Some examples of structural patterns are *Trusted Proxy* and *Separate Security Layer*.

Procedural patterns use structural patterns to define an approach to a problem. The difference is often in the wording. A procedural pattern name will be some imperative. If the word "Do" in front of a pattern name makes sense, it is a procedural pattern. An example of a procedural pattern is *Build From the Ground Up* [Kienzle02].

The Build From the Ground Up Pattern

Build From the Ground Up

Name:

Build From the Ground Up

Alias:

Start From Scratch

Know Your System

Intent:

To ensure that the owners and maintainers of a secure system understand how their system works.

Motivation:

Software application loopholes tend to be the most readily exploited vulnerability on a network [Yoder97]. Often, the network administrators are unaware of the loopholes. A complex system is dangerous because it is difficult to monitor the weak links. It becomes impossible if the system architects don't even know that those weak links exist. If a system must be complex, which it often must be in order to support business needs, then it is imperative that the architects know every service that is running, every piece of hardware attached, every application, every protocol, every open port, and every way in and out of the system.

Principles:

Secure the Weakest Link
Keep It Simple
Principle of Least Privilege

Applicability:

Follow this pattern when you are building a new network, or overhauling an existing one with new hardware or software, particularly when migrating to a new operating system or hardware platform.

Forces:

- Attacker
  - He wants into your system. He will often know more about the security signatures of the hardware and software that you're running than you do.
- Enclave
  - The protected system you maintain or are attempting to build. This pattern inherits the Enclave pattern's forces. The elements of the enclave (Guard, Access Policy, etc) represent the software that will govern your system.
- A complex system is more difficult to secure
- Operating System and Server software is often rolled out with a default implementation that works right out of the box.

Solution:

The solution has four parts [Yoder97].

1. Understand the default implementation. A lot of operating systems and web servers are designed to work right out of the box with minimal configuration on the part of the user. This is mighty convenient unless you are at all concerned with security. That isn't to say that you should never use the default plug-and-play settings. Just

know what they are. A good example of this is building a Red Hat Linux server. The graphical tool provided is extraordinary. It allows you to disable or enable options in groups or individually. The shocking thing is just how much is enabled by default.

2. Simplify the configuration. Don't allow yourself to be overwhelmed by the configuration options. You may find yourself selecting options you don't even understand, simply out of frustration. There are two keys. First, let the problem domain dictate your actions. Focus on the settings that you know you need in order to meet your design objectives. Second, document everything. Keep a record of all the software you install, including version numbers, support and download web addresses, and the date you installed it. That way, when you go to rebuild a system you won't become overwhelmed.
3. Remove any services that you do not need. The less you have running on your server, the less you have to keep track of.
4. Investigate vulnerabilities that may be built into your software. Keep track of security-related news, particularly for the software that you use. Subscribe to mailing lists from the relevant vendors for this purpose. Visit http://www.sans.org/newsletters/ for current security news. "Keep it simple" will only get you so far. There comes a point when you just have to know the code.

Consequences:

- Makes for a longer time to market. It will take a lot of time up front, especially if you are just learning the ins and outs of the hardware and software you use to implement.
- You will save time in the long run if you don't have to constantly keep track of vulnerabilities in software and services that you don't use.

Related Patterns:

The Enclave Pattern is the net result of a successful implementation of this pattern.

The Separate Security Layer Pattern

Separate Security Layer

Name:

Separate Security Layer

Alias:

Bottom Up Security

Intent:

Ensuring that an application's security mechanisms are not undermined by an insecure framework, and providing a loose coupling between application and security.

Motivation:

Security is an aspect of any software system that needs to be flexible and able to change with the times. Applications that provide their own security mechanisms are difficult to maintain. If the security implementation turns out to be faulty, the whole application must be refactored. New applications take much longer to write because they too must implement some security. Often, security will be implemented the same way in each new application. This is especially the case in the web domain. This redundancy can lead to the duplication of errors, and can be a maintenance nightmare.

Furthermore, sensitive authentication data, like passwords, is only as safe as the application is, and often, applications are not designed with security in mind.

Principles:

Defense In Depth

Secure the Weakest Link

Applicability:

Use this pattern when you are designing a suite of software applications that need to share a security implementation, or if you intend to write the security implementation upon which some applications may run. This pattern allows you to move forward with design even before the details of your security implementation are nailed down if you consider security functionality to be a *hinge point* or area of design that will likely change.

Forces:

- Framework
  - The foundation upon which your application is built
- Application
- An application that interacts with low-level security mechanisms is difficult to debug and impossible to transport.
- Interfacing with an external security solution is not always possible

Solution:

Use the security mechanisms provided by your application framework. If there isn't one, design one separate from your applications that can run underneath them. Often a database will be involved in your security implementation to store login ids and passwords, roles, etc. Your applications should never interact directly with the security database.

Instead, your security framework should do it and provide a separate connection to your applications, wherein the application requests authentication *through* the security layer.

On a smaller system, the same database that is used for application data is used for security. This is okay, but you should always design your security system as if it had its own database. Using the same connection mechanism for your security layer that is used by your applications causes an unnecessary coupling between the systems. Provide a distinct connection mechanism for your security layer [Probst02].

Consequences:

- Any new application developed can hook into an existing security framework
- Old applications must be refactored with security-related code removed.
- Your security framework will become a single point of failure for all your applications (so make sure it's secure).

Related Patterns:

The Distinct Connection mini-pattern [Probst02] supports the decoupling of security implementation from applications.

Conclusion: The Design From a Distance Pattern

A good design starts with principles and objectives. Its purpose never strays from the adherence to those principles, nor from the fulfillment of those objectives. It's easy to become distracted by the sheer volume of information and tools available to the security professional, but the most valuable tool that you'll ever have is a mind that can identify patterns in chaos, and overcome complexity by abstraction. Remember, the Devil's in the details, so always design from a distance.

Design From a Distance

Name:

Design From a Distance

Alias:

None

Intent:

Separating the design criteria from the details of the implementation

Motivation:

Today's corporate systems are complex. The human mind was not designed to track the quantity of details that must be tracked in order to securely maintain these large systems. That's why the human mind instinctively groups objects together and seeks patterns in chaos.

Principles:

Keep it Simple

Compartmentalize

Applicability:

Use this pattern whenever you begin to feel overwhelmed by the task of designing and implementing a security policy for your network or system.

Forces:

- The human mind is limited in the number of details it can track
- Abstract problem and solution pairs tend to be recurrent across all operating systems, languages, and applications.
- Every complicated information security paradigm has a real-world analogy.

Solution:

In information security, the beginning of wisdom is to know what you do not know. Design patterns are the result of many years of mistakes and successes. They capture the experience and knowledge of the best in the industry. If you are just getting started, there is no need to get wrapped up in the gory details. Pattern catalogs are the playbooks of the experts. Use them.

Let the problem domain dictate your course of action [Hohm03]. In other words don't "goldplate." Keep your solution as simple as it can be while meeting the requirements. Also, your organization should have an understanding of the real value of the assets you are protecting. In some cases, cost-benefit analysis will prevent unneeded expense in developing a security solution. Security design without assessment of the business value is an anti-pattern that should be avoided [Miroslav02].

Keep an eye on the big picture. The most complicated system is always just a system of patterns. Learn to recognize them. They may overlap. In fact a good patterns should overlap according to Alexander, the Father of Design Patterns [Alexander79].

If you are an IT professional, your knowledge of security will be tested. It is inevitable. Fortunately, when it is, you can stand on the shoulders of giants, and be confident that the solutions you recommend are effective ones. Patterns make it possible.

Consequences:

- Avoid burnout

- Avoid costly mistakes that have already been made and documented a million times

Related Patterns:

All patterns are related in that this pattern acts as a guide for implementing security patterns in general.

List of References

[Alexander79] Alexander, C. *The Timeless Way of Building*. Oxford University Press, 1979.

[GoF94] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hillside] Hillside.net: Your Patterns Library, http://www.hillside.net/patterns

[Hohmann03] Hohmann, Luke, *Beyond Software Architecture*, Addison-Wesley, 2003.

[Kienzle02] Kienzle, Darrell, Ph.D and Matthew Elder Ph.D., *Security Patterns For Web Application Development*, 2002, http://www.scrypt.net/~celer/securitypatterns/ final report.pdf

[Miroslav02] Miroslav Kis, Ph.D., *Information Security Antipatterns in Software Requirements Engineering*, 2002, jerry.cs.uiuc.edu/~plop/plop2002/ final/mkis_plop_2002.pdf

[OG] The Open Group, http://www.opengroup.org

[OG02] The Open Group. *Guide to Security Patterns.* http://www.opengroup.org/security/gsp.htm, 2002.

[Probst02] Probst, Stefan, *Reusable Components for Developing Security Aware Applications*, 2002, http://www.acsac.org/2002/papers/88.pdf

[Saltzer75] Saltzer, Jerome and Michael Schroeder, *The Protection of Information in Computer Systems*, 1975, http://www.cs.virginia.edu/~evans/cs551/saltzer/

[Schumacher01] Schumacher, M. and U. Roedig. *Security Engineering with Patterns*, Monticello, IL, 2001.

[SecPat] Security Patterns Homepage, http://www.securitypatterns.org/

[Sturgeon03] Sturgeon, Will, "What's the Biggest Threat to Your Network?", 2003, http://www.zdnet.com.au/news/business/0,39023166,20273295,00.htm

[Yoder97] Yoder, J. and J. Barcalow, Architectural Patterns for Enabling Application Security, 1997, http://www.joeyoder.com/papers/patterns/Security/appsec.pdf