Security Patterns
Template and Tutorial

Darrell M. Kienzle, Ph.D.
Matthew C. Elder, Ph.D.
David S. Tyree
James Edwards-Hewitt

Introduction

There is a huge disconnect between security professionals and systems developers. Security professionals are primarily concerned with the security of a system, while developers are primarily concerned with building a system that works. While security is one of the non-functional goals that developers must be concerned with, it is but one of many. And while security professionals complain that developers don't take security seriously, developers are just as frustrated that security professionals don't understand that security is not their only concern.

Security patterns are proposed as a means of bridging this gap. Security patterns are intended to capture security expertise in the form of worked solutions to recurring problems. Security patterns are intended to be used and understood by developers who are not security professionals. While the emphasis is on security, these patterns capture the strengths and weaknesses of different approaches in order to allow developers to make informed trade-off decisions between security and other goals.

Above all, security patterns are meant to be constructive. Far too much of the available security expertise is presented in the form of laundry lists of what not to do. The poor developer who attempts to understand security is likely to be overwhelmed by these lists. Security patterns instead try to provide constructive assistance in the form of worked solutions and the guidance to apply them properly.

This document presents our template for developing and presenting security patterns. It is intended for the potential author of security patterns. We first discuss the notion of a pattern, and then the application of patterns to security. Next, we explore the different abstraction levels possible and its relationship to the target audience. We then present our security patterns template and provide guidance in writing security patterns. Finally, we outline related work in this space.

What Is a Pattern?

One of the most exciting developments in software engineering is the emergence of Design Patterns as an approach to capturing, reusing, and teaching software design expertise. This movement first gained widespread visibility with the publication of *Design Patterns* [GAMM] in 1994. The "Gang of Four book", as it is commonly known, defined design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

The patterns approach has infiltrated all areas of software engineering. For example, the Java APIs, the OpenStep libraries, and the Microsoft Foundation Classes all use the patterns catalogued in Design Patterns. There are numerous workshops, books, and web sites devoted to the further study and development of design patterns.

The design patterns movement actually has its roots outside of software engineering entirely. Christopher Alexander invented the concept in his writings on architecture and urban planning. He developed the approach in order to capture the essential knowledge of his field, and provide a methodology in place of ad hoc craftsmanship. Alexander's patterns ranged from high level (placement of buildings within a community) to low level (the layout of an individual room). An excellent discussion of Alexander's work can be found in Gabriel's Patterns of Software [GABR]. Gabriel's book should be required reading for the design patterns community in that it presents a balanced view of both the successes and the failures of Alexander's approach.

Alexander defined a pattern as describing "a problem which occurs over and over again in our environment, and that 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". Vlissides paraphrased Alexander's definition of a pattern as "a solution to a problem in a context". Vlissides posited that in addition to the problem, context, and solution, a pattern must include recurrence, a teaching component, and a name.

Alexandrian patterns generally include the following major elements:

- A standard name by which the pattern can be discussed
- A concise summary of the problem addressed by the pattern
- A description of the solution
- The impact that the pattern has on the “essential forces” at play

Design patterns generally adhere to a much more restrictive format. They include UML diagrams of the structure and the dynamic interactions of the objects that constitute the pattern. They require examples of the patterns in use. They include pointers to related patterns. And they generally include sample code implementing the pattern.

The design patterns approach has been very successful at capturing the recurring code-level patterns that occur in object-oriented software. There is a large community developing exactly this type of pattern. However there are at least three significant variations worth mentioning:

- *Architectural patterns*. Peter Coad recognized that many software design patterns could actually be applied at higher levels of abstraction. For example, a publish-subscribe model described in a design pattern could actually be used at the enterprise level between entire systems. Coad developed a repository of “architectural patterns” to capture this. While similar in nature to design patterns, this work demonstrated the general utility of Alexander’s patterns approach applied to software.
- *AntiPatterns*. Another group of four developed “AntiPatterns” as an approach to documenting common mistakes in software development [BROW1]. These AntiPatterns document common implementation, architecture, and even software engineering process mistakes. The AntiPatterns books claim to provide mechanisms for “refactoring” inappropriate approaches into more acceptable alternatives. However, the major focus of the AntiPatterns approach is in enumerating failures.
- *Pattern Languages*. Alexander actually developed the notion of a pattern language as a mechanism for expressing collections of interrelated patterns. Using a pattern language, it is possible to define families of solutions that allow plug-replacement of specialized components. This approach is quite popular, as it lends itself well to the development of object-oriented frameworks of interrelated components.

What Is a Security Pattern?

A security pattern is a well-understood solution to a recurring information security problem. They are patterns in the sense originally defined by Christopher Alexander applied to the domain of information security. While some of these patterns will take the form of design patterns, *not all security patterns are design patterns*.

Because of the popularity of design patterns in the software engineering community, the natural inclination is to assume that anything going by the name “security patterns” should be described using a UML diagram and include sample source code. While it is true that many interesting security patterns can be presented this way, there are many other important patterns (some procedural, some architectural) that do not fit within these constraints.

We make no attempt to categorize formally different classes of pattern. We have observed a few broad types, but we don’t feel it important to rigidly enforce any formal identification. For informational purposes, we identify two broad categories:

- *Structural patterns*. These are patterns that can be implemented in the final product. They encompass design patterns such as those used by the Gang of Four. They usually include diagrams of structure and dynamic interaction.
- *Procedural patterns*. These are patterns that can be used to improve the process for development of security-critical software. They often impact the organization or reporting structure of the project.

Our patterns adhere to our patterns template as described in a later section of this document. We do not claim that this template is perfect or complete. We started the project with a general idea of what should be in the template. Over time, the template has been refined as fields were added, eliminated, or combined. We expect that this refinement will continue as we learn more. Nevertheless, we feel that it is important that our patterns follow some basic structure, conveying the same basic information and answering the same basic questions.

It is important to note that there are a number of different efforts bearing the name “security patterns”. Please see the section on *Related Work* for a discussion of other approaches to (and other definitions of) security patterns. The remainder of this section will discuss our definition of security pattern. Differences will be discussed under *Related Work*.

Levels of Detail / Target Audience

There is no single correct level of detail for security patterns. Different potential consumers of security patterns work at different levels. A developer may be primarily concerned with patterns of code-level objects, an architect may build network models, and a CIO may be primarily interested in trust relationships between organizations. All are valid uses of the security patterns approach, though each target audience might find little of value in patterns at a much different level of detail.

We have found it useful to distinguish between four different categories of pattern detail:

1. *Concepts*. This is the highest level of abstraction, and encompasses general strategies such as diversity, obscurity, and least privilege. These are abstract nouns and cannot be implemented directly by developers. For example, one cannot build a “Least Privilege”.

2. *Classes of patterns* (also called pattern families). A class of pattern represents a general problem area for which multiple solutions are possible. While a class of pattern cannot be directly implemented, it can be used as a placeholder for tradeoff analysis between alternative approaches.

3. *Patterns*. A pattern is specific enough to allow basic properties to be specified and trade-off analysis to be conducted against other patterns. It is general enough to allow it to be used in multiple circumstances with multiple targets.

4. *Examples*. An example is a worked solution to a very specific problem instance. An example is typified by sample code. It is the most immediately useful, but in a very narrow context.

As an example, consider the desire for secret communication between two parties. An executive may identify the need for confidentiality. An architect could suggest an encrypted communication as a class of pattern. The software designer could select a session encryption pattern supported by an out-of-band authentication pattern. Finally, a developer would instantiate the pattern using specific algorithms, libraries, source code, and key strengths.

We must reiterate that this is an arbitrary selection of levels of abstraction. An architect might refer to level 2 as “patterns” and believe that level 3 and below represent nonessential details.

Alternatively, the businessman might see levels 2, 3, and 4 as all nonessential details that the technical types can deal with. The implementor might see level 3 and above as hopelessly abstract. In all cases, people are most comfortable working at a certain level of abstraction.

We have selected level 3 for our patterns (with examples from level 4) because we believe it represents the point where we can get the most benefit in terms of transfer of security knowledge. Any more concrete and we run the risk of losing the message amidst the details. Any more abstract, and we lose interest of the developers who, ultimately, build the software upon which we depend.

The Security Pattern Template

A security pattern consists of the following named elements. Wherever possible, we have cross-referenced our element titles with other patterns templates, such as the Gang of Four or AG Communication Systems (AGCS). Some of these elements might not be applicable in all pattern examples; in such cases, the field should be labeled "N/A".

Pattern Name

The *Pattern Name* should capture the essence of the pattern in a concise and, if possible, catchy manner. Avoid using terms that already have strong connotations.

If possible, structural patterns should be nouns that describe the thing that can be built, e.g., "Partitioned Application" or "Trusted Proxy". More abstract nouns are appropriate when the pattern is itself more abstract. For example, the "Password Authentication" pattern is a general pattern that describes many facets of password authentication mechanisms.

Procedural patterns should be in the form of active verb phrases that describe the action the pattern recommends. For example, "Proactive Patching" was amended to "Patch Proactively".

Abstract

The Abstract should summarize the pattern briefly in two to three sentences. The summary should include what the purpose or intent of the pattern is. It should not go into implementation details.

When developing the abstract, consider how it will be used. Users of the security patterns repository who perform a query on the database (e.g., a keyword search) will be provided with a list of pattern names and abstracts. The abstracts must stand completely independent of any context, and they must be concise.

Abstracts will allow users to decide which patterns to explore in greater detail. Thus, they should give some indication of any limits on pattern applicability. If, for example, a pattern should not be used to protect high value data, that would be something to note in the abstract.

The Abstract is related to the Gang of Four's *Intent*.

Aliases

The *Aliases* should enumerate other names for the pattern, including names by which others might have referenced the pattern in the literature.

The *Aliases* are equivalent to the Gang of Four's *Also Known As*.

Problem

The Problem should describe the conditions that motivate the usage of the pattern. This section outlines the context in which the pattern is applicable, as well as explaining the motivation for using the pattern. When multiple patterns address the same basic problem, the Problem for each pattern should provide the more detailed context that would make that pattern specifically appropriate.

The problem statement should not contain a lengthy discussion of secondary effects. For example, the Problem solved by the password authentication pattern does not include the need to protect against password guessing attacks. The password authentication pattern addresses the problem of authenticating users. Susceptibility of this approach to password guessing attacks is a secondary effect of using passwords.

The Problem is related to the Gang of Four's *Motivation* and the AGCS' *Context*.

Solution

The *Solution* should describe at a high level how the pattern solves the problem described in the problem statement. This section should explain how the pattern is applicable to the problem and the rationale for applying the solution.

The *Solution* is related to the Gang of Four's *Applicability* and the AGCS' *Rationale*.

Static Structure

The *Static Structure* should present the constituent elements involved in the usage of this pattern.

For structural patterns, a *Diagram* should be used to depict the architecture visually. For object-level patterns, UML could be used. For other structural patterns, any graphical notation that enhances the communication of the pattern can be used. For some structural patterns, a diagram simply may not be appropriate. In these cases, this fact should be noted.

For procedural patterns, the diagram could represent the actors in the process being described. For some procedural patterns, a diagram is not applicable.

When a diagram is presented, it should be accompanied by an enumeration of the *Components* with a detailed description of each.

The list of *Components* is equivalent to the Gang of Four's *Participants.*

Dynamic Structure

The *Dynamic Structure* should outline the interactions between the various components in the static structure. This section should present each of the significant collaboration scenarios that are included in the pattern.

The *Dynamic Structure* is similar to the Gang of Four's *Collaborations.*

Implementation Issues

The *Implementation Issues* should provide some wisdom in the form of detailed hints and techniques. It is important to not overwhelm the reader. The most significant issues should be addressed first, and in the most detail. Minutiae should be given relatively little attention.

This section should identify the most common mistakes in the usage of this pattern (pitfalls) and provide the reader with guidance for avoiding them.

This section can also state explicitly how the attacks identified in the next section, *Common Attacks*, could be avoided through the use of this pattern. (Although this should be done sparingly, as it creates a "forward reference" problem that we have yet to resolve in our template.)

Common Attacks

The *Common Attacks* should identify attacks that interact with this pattern. Where possible, links to databases such as bugtraq, securityfocus, or CVE should be provided,

This section should also enumerate residual vulnerabilities by explaining circumstances wherein a properly implemented pattern might still be attacked.

Known Uses

The *Known Uses* should cite examples of this pattern that are known to be in actual use. Explicit references to products or systems can be used. However, in recognition that many systems do not want their security internals discussed, it is appropriate to refer to "a known system" without identifying that system. Likewise, it may be necessary to provide only general details of a specific system.

Known Uses should be identified as occurring at one of three levels:

Code level. This level involves object interactions and is similar to the material presented in the Gang of Four book. From a security perspective, it often relies on the protection features of the programming language.

System level. This level involves interactions between independently developed products and separate processes. These often rely on the security features of the operating system.

Network level. This level describes system interactions distributed across a network. Security mechanisms can be implemented using firewalls and other network-level components.

Although many patterns will target a specific level, examples from all three levels should be included wherever possible in order to communicate the generality of a pattern.

Sample Code

Developers appreciate sample code that they can link directly into their application and begin using immediately. Although these patterns are not a sourcebook, it is important to present samples whenever possible. They make the material more tangible and gain credibility with the intended readership.

Consequences

The *Consequences* should describe the possible impact of using the pattern with respect to various functional and non-functional requirements. This section hypothesizes about the competing forces that come into play in the usage of the pattern.

Each of the following areas of potential consequence should be discussed. If the pattern appears to have no impact on the consequence, indicate "None."

Accountability	Accountability is the process of tracing information system activities to a responsible source [NSTI]. How does the pattern impact the ability to hold users accountable for their actions? What effect does system compromise have on accountability? This requirement includes issues in logging, control, audit, authentication, and non-repudiation.
Confidentiality	Confidentiality is the assurance that information is not disclosed to unauthorized persons, processes, or devices [NSTI]. How does the pattern impact confidentiality of a single user's data as well as the system as a whole? What effect does system compromise have on confidentiality?
Integrity	Integrity is the assurance that information is not modified or destroyed in an unauthorized manner [NSTI]. How does the pattern impact integrity of a single user's data as well as the system as a whole? What effect does system compromise have on integrity?
Availability	Availability is the timely, reliable access to data and service for authorized users [NSTI]. How does the pattern impact the availability of the system to a single user, a group of users, or all users? What effect does system compromise have on availability? Include susceptibility/defenses against denial of service attacks.
Performance	How does the pattern affect system performance? Does it alter the load that the system can handle without degradation? Does it consume bandwidth, cycles, or other resources?
Cost	How expensive is the pattern to implement? Does it add to development cost? Are there operational costs associated with it? Does it require more bandwidth, cycles, disks, machines to implement?

Manageability	Does the pattern introduce or reduce a management burden? This includes impact on maintenance.
Usability	Does the pattern detract from (enhance) the overall user experience? Keep in mind that the target audience for most web sites is not sophisticated users.

The *Consequences* are related to AGCS' *Forces*.

Related Patterns

The *Related Patterns* should list and include links to any other related patterns in the repository. Provide a brief description as to the nature of this relationship. Two patterns could be related in any number of ways: one pattern could use another, one pattern could be dependent upon another, one pattern could replace another, etc.

If related security patterns from other sources are known, indicate them here.

References

The *References* should enumerate citations related to the pattern in the literature and on the web. These references should be cited throughout the pattern description.

Related Work

This document provides our definition of a security pattern, and our template for documenting security patterns. There are other groups working in the same general space, attempting to apply the concepts of Design Patterns to security. There is no "one-size-fits-all" solution. This section presents a very brief overview of other work so that the reader can best determine which approach will best address his/her specific needs.

Security Properties of Design Patterns

The very first reference to design patterns in a security context was a presentation by Deborah Frincke at NISSC 1996. The presentation discussed the security ramifications of some of the original 23 Design Patterns [GAMM]. Her basic idea was that one could start with a security requirement on a pattern, and decompose it into constituent object requirements. In this way, if a proof were developed at the pattern level, it could be reused in tandem with the pattern. This was a powerful idea, but limited to patterns that were not originally intended to address security requirements. Unfortunately, this work was not continued, and was never published.

The NRL Patterns Work

The Naval Research Lab is currently involved in applying patterns to the problem of formal verification of security-critical software. Based on early work using claim trees as an assurance methodology, the researchers at NRL believe that claim trees could be made more cost-effective through reuse in tandem with common patterns of software architecture. An assurance professional could create a generic argument to accompany a software pattern. Software developers could then instantiate that pattern in circumstances that require the functional and assurance properties that it provides.

The NRL patterns are "heavyweight," in that they are costly to develop, being geared at automated analysis and manipulation. In contrast, our patterns are "lightweight," as they are aimed at communication with practitioners. While NRL focuses on how patterns should be best represented, the primary thrust of our research is to collect the correct patterns. Ultimately, it is conceivable that the two efforts can be merged, with the structural patterns from our repository being encoded using the notation that NRL develops.

Security Patterns at PLOPD

The annual Pattern Languages of Program Design (PLOPD) workshop brings together patterns researchers working in many different domains. Two papers have been presented at PLOPD that discuss the use of patterns to address security requirements. Both of these are very low-level approaches, providing techniques for developing object-oriented software that is flexible with regard to security. The first such paper presents an object-oriented abstraction of existing Cryptographic APIs [BRAG]. Using these patterns, a developer can construct software that can easily be ported from one Cryptographic API to another. The second paper presents developers with strategies for leaving security placeholders within code, so that basic access control mechanisms can be added at a later date [YODE]. These papers are important in that they introduce the idea of security-specific patterns. However, they do not really raise the user's awareness of the pitfalls and trade-offs that are present.

The Security Patterns mailing list

Over the past year, a small group of researchers working in this space has used a sporadic mailing list to share their work. Details of the mailing list can be found at www.security-patterns.de. The individual researchers have developed a number of patterns, largely in isolation. The web site has a fairly comprehensive list of links in this area.

Most of the patterns presented on the mailing list have been conventional design patterns, and have used a template similar to the Gang of Four. A few, however, have been procedural patterns.

The OpenGroup Security Forum

The OpenGroup Security Forum is currently developing a library of architectural security patterns. We have seen them but aren't at liberty to discuss them. We can say that the patterns they are developing are essentially what we have termed "pattern families."

Ultimately, our patterns may serve as concrete examples of their patterns, and their patterns may provide precise statements of need for our patterns. But at this early stage it would be premature to overly constrain either effort by trying to force them to align.

Conclusion

Security Patterns are a novel technique for encapsulating and disseminating security expertise. In this paper we have presented our template for security patterns and provided a tutorial for our approach to writing security patterns.

References

General Security

[NSTI] NSTISSC (National Security Telecommunications and Information Systems Security Committee). National Information Systems Security (Infosec) Glossary. NSTISSI No. 4009, September 2000.

General Patterns

[HILL] Hillside, The Hillside Web Patterns Repository ???, http://www.hillside.net.

[ALEX] Alexander, C., S. Ishikawa, M. Silverstein, et al. A Pattern Language. Oxford University Press, New York, 1977.

[BROW1] Brown, W., R Malveau, H. McCormick, and T. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, 1998.

[GABR] Gabriel, R. Patterns of Software: Tales from the Software Community, 1997.

[GAMM] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[VLIS] Vlissides, J. Pattern Hatching: Design Patterns Applied. Addison-Wesley, 1998.

Security Patterns

[BRAG] Braga, A., C. Rubira, and R. Dahab. “Tropyc: A Pattern Language for Cryptographic Software”. Pattern Languages of Programs 1998, Monticello, IL, 1998.

[BROW2] Brown, F., J. DiVietri, G. Villegas, and E. Fernandez. “The Authenticator Pattern”. Pattern Languages of Programs 1999, Monticello, IL, 1999.

[FERN1] Fernandez, E. “Metadata and Authorization Patterns”. Technical report TR-CSE-00-16, Computer Science and Engineering Department, Florida Atlantic University, 2000.

[FERN2] Fernandez, E. and R. Pan. “A Pattern Language for Security Models”. Pattern Languages of Programs 2001, Monticello, IL, 2001.

[HAYS] Hays, V., M. Loutrel, and E. Fernandez. “The Object Filter and Access Control Framework”. Pattern Languages of Programs 2000, Monticello, IL, 2000.

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

[SCHU] Schumacher, M. and U. Roedig. “Security Engineering with Patterns”. Pattern Languages of Programs 2001, Monticello, IL, 2001.

[SECU] Security Patterns Mailing List, http://www.security-patterns.de

[YODE] Yoder, J. and J. Barcalow. “Architectural Patterns for Enabling Application Security”. Pattern Languages of Programs 1997, Monticello, IL, 1998.

Appendix A – An Example Security Pattern

Network Address Blacklist

Abstract

A Network Address Blacklist prevents further access to a system by misbehaving clients. Sometimes it is not possible to determine a single client user account that is attacking the system. In these circumstances, the only recourse of the site administrator is to blacklist the network address.

Aliases

- Client Filtering
- IP Lockout

Problem

In a web-based environment, where anonymous access is possible, there can be little to prevent an attacker from brazenly attacking your system, even after being detected. Unless you are a commercial bank, the FBI is too busy to track down hacker attacks on your web site. When deterrence fails, you need some means of defending your site against anonymous attackers. It is not sufficient to simply rely on the strength of your static defenses. Attackers that probe your site and find no resistance are likely to become more brazen.

Many systems implement account lockout if too many incorrect login attempts are made against a particular account. This can be a double-edged sword because an attacker can launch a massive denial of service by randomly guessing several passwords for many different accounts. This may be pure mischief, or it may have the intent of masking a guessing attack on a specific targeted account. Alternately, an attacker could choose a single weak password and then conduct an "account guessing attack" until an account using that password is discovered. The lockout mechanism would never detect such an attack.

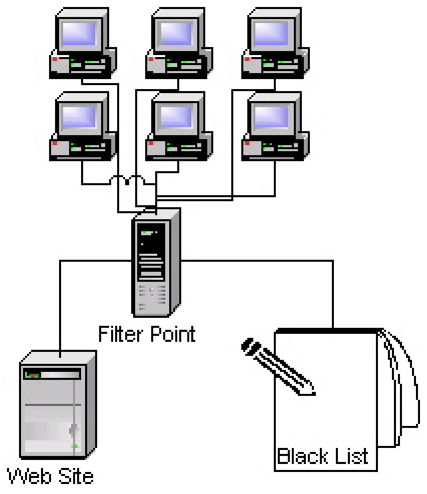
Solution

A Network Address Blacklist mechanism maintains a list of network addresses that have exhibited inappropriate behavior. Once a given network address has exhibited enough such behavior to have obviously malicious intent, the network address will be temporarily added to a blacklist. When a request is received from a blacklisted address, it will be dropped on the floor or redirected to a page informing the attacker that his/her activities are being monitored and will be reported to the authorities.

All web accesses are based upon TCP, which uses a handshake protocol to ensure that incoming IP addresses are not spoofed. While simple denial-of-service attacks can be spoofed, more sophisticated HTTP-based malfeasance requires that the attacker be able to receive responses at the originating address. As a result, most attackers will have only have a handful of different IP addresses at their disposal. Blocking requests from a questionable IP address can effectively throttle many automated attacks.

Static Structure

Diagram



Components

- Client (characterized by an originating IP address)
- Filtering point that receives network addresses and compares them to a blacklist
- The blacklist of network addresses that are currently locked out
- A series of sensors that collect suspicious behavior and report it to the administrator
- An administrator or administrative module that collects reports of misbehavior and makes the decision to blacklist. The administrator may also have a direct relationship with the client.

Dynamic Structure

Interaction 1: A normal client makes a normal request

Interaction 2: A normal client makes a suspicious request

Interaction 3: A blacklisted client makes a request of any sort

Interaction 4: The administrator removes the client from the blacklist (possibly at the client's behest)

Implementation Issues

- Do you block at the firewall, by the web server access control rules, or at the application? The further forward, the more effective (less resources consumed). The further back, the simpler to implement.
- Do you maintain the list of addresses statically or dynamically? Dynamic is more responsive, but can add complex (possibly dangerous) APIs between components. Also, you don't want the firewall querying the database on every access. Static is simple to update periodically but not very responsive -- if the we server needs to be restarted to reload the list, this could be painfully unresponsive.
- When you block a network address, you run the risk of blocking access to an entire organization behind a NAT firewall. This is actually a feature. If legitimate users within the organization complain about being unable to reach your site, they will be in a position to track down the misbehaving party, and may even punish that individual.
- Do you block and tell the user they are blocked or simply drop incoming requests at the firewall? If you block at the firewall, automated attack tools will be completely blunted. Telling the user that they are blocked gives them information about your security policy, and consumes resources. Plus, attack tools may continue trying COTS vulnerabilities, and may eventually hit on one.
- How long should the network address be blacklisted for? You need to blacklist and address long enough to be effective and slow a network attack down, but not so long that a legitimate client will suffer if they make a mistake.
- You must have an interface to manage the blacklist, so that if a client contacts you about being unable to access the site, you can rectify the problem.
- Do you establish an automated blacklist mechanism, or do you rely on a human administrator to administer the blacklist? An automated mechanism can be more responsive, but a manual approach may be easier to integrate. Ultimately, the decision

may be made by policy -- most shared firewalls cannot be manipulated by a single web server.

Some of the sensors that can be used to determine misuse by a network address are:

- When a single network address generates a number of requests for non-existent pages, they may be performing an automated search of your system.
- If your site does not use cgi-bin scripts, any attempt to request a cgi-bin script will not have come from a legitimate user clicking on a link. Many such requests indicate that an off-the-shelf web vulnerability scan is being performed.
- If multiple invalid login attempts are received from the same network address, it may be a user who is attempting to guess usernames and passwords. Likewise, if a reserved name such as "guest", "root", or "administrator" is used, the client is likely up to no good. However, it is possible that a legitimate user has simply forgotten their username. In this case, the threshold for invalid attempts should be set quite high.
- If the server receives requests containing invalid data that should have been filtered by client side validity checks. This is often an indicator that the user is deliberately bypassing those checks in hope that the server will act on the incorrect data. See *Untrusted Client.* It is important to recognize that in some cases -- for instance, when Javascript is disabled on the client or blocked by the client's web server -- the client side checks may be disabled through no fault of the user. If the site must be accessible to users who will not accept Javascript, this should not be considered as evidence of attempted misuse of the site.

Common Attacks

- Denial-of-Service Attack - the attacker may constantly hit your site after he/she has been blocked causing your site to eat up resources to verify their address, or the attacker may purposely block addresses so that legitimate clients sharing the same address cannot access the site.
- This pattern may be ineffective against large scale DDoS attacks where many individual network addresses are used.
- Attackers may make it appear that the attack is coming from critical infrastructure such as your router, database, or firewall, causing you to block these servers, rendering your site in-operable.

- Attackers may grow the blacklist to a large size so that it is inefficient to examine the blacklist, causing a general degradation in performance.

Known Uses

Code Level

N/A

System Level

N/A

Network Level

- Maps RBL/RSS - Maps is a service which blocks spam by IP. Spammers are resisted with the Maps service and then when incoming mail or web request arrive the servers using the system will check the IP address to see if it is on these lists. If so the mail or request is rejected. See http://mail-abuse.org/ or http://www.thestandard.com/article/display/0,1151,2889,00.html
- Many commercial IDS such as ISS RealSecure offer features and functionality that block attackers in real-time. See http://www.realsecure.com
- We have personally seen two web sites that implement this sort of mechanism but are not at liberty to discuss particulars.

Sample Code

TBD

Consequences

Accountability	The network address blacklist is necessary when there is no other recourse, and by definition no accountability. In some sense, denying further access is a form of punishment and could be considered a measure of accountability.
Confidentiality	When implemented effectively, a Network Address Blacklist enhances site confidentiality and integrity. This occurs because it will dissuade or prevent attempts to misuse the web application (since many forms of misuse are intended to compromise those security characteristics). Even if ineffective, it will not adversely affect these properties.
Integrity	See Confidentiality.
Availability	A Network Address Blacklist can have a negative effect on availability. Two specific cases are: • The Network Address Blacklist offers an interface whereby specific network addresses can be blocked from further access. If this interface is misused or implemented incorrectly, it could cause legitimate users to be denied access. • All users whose requests originate from behind the same firewall may appear as if coming from the same network address. It is conceivable that one user could cause an entire enterprise to become locked out of the web site.
Performance	The Network Address Blacklist has mixed effects on performance. On the one hand, misbehaving users who might otherwise be consuming considerable resources are prevented from gaining access to the site. On the other hand, the lockout mechanism could itself incur significant performance penalty on all users. If a Network Address Blacklist is implemented in a way that accesses a database for every web request, the performance will suffer. If the lockout depends on an in-memory blacklist, it is important that the size of the list be bounded and that the list does not itself become a bottleneck.
Cost	If lockout can be implemented using existing APIs, it can be developed quite cheaply. If novel interfaces are required, it will be quite expensive to develop. In either case, quality assurance will be non-trivial.
Manageability	A Network Address Blacklist can have a positive effect on the administrative burden of the web site. By eliminating many nuisance attacks, the sysadmin can be freed to investigate more significant security concerns. However, if legitimate users are finding themselves locked out on a routine basis, the management burden will be raised. Particularly if administrators have to manually intervene to clear an address from the blacklist.
Usability	As long as the misuse sensors are not too finely attuned, legitimate users will never see any impact on usability. Those who attempt to misuse the system will see a significant negative effect on their usability of the system

Related Patterns

- Untrusted Client

- Password Authentication

References

1. Brown University Computing and Information Services. "IP Address Filtering", *Web Authorization/Authentication*. http://www.brown.edu/Facilities/CIS/ATGTest/Infrastructure/Web_Access_Control/GoalsOptions-ver2.html#anchor136476, October 1997.
2. Stein, L. and J. Stewart. "Protecting Confidential Documents at Your Site", *The World Wide Web Security FAQ*. http://www.w3.org/Security/Faq/wwwsf5.html#CON-Q2, February 2002.