# Attack Patterns as a Knowledge Resource for Building Secure Software

**Sean Barnum**
**Cigital, Inc.**

**Amit Sethi**
**Cigital, Inc.**

*Building software with an adequate level of security assurance for its mission becomes more and more challenging every day as the size, complexity, and tempo of software creation increases and the number and the skill level of attackers continues to grow. These factors each exacerbate the issue that, to build secure software, builders must ensure that they have protected every relevant potential vulnerability; yet, to attack software, attackers often have to find and exploit only a single exposed vulnerability. To identify and mitigate relevant vulnerabilities in software, the development community needs more than just good software engineering and analytical practices, a solid grasp of software security features, and a powerful set of tools. All of these things are necessary but not sufficient. To be effective, the community needs to think outside of the box and to have a* ***firm grasp of the attacker's perspective and the approaches used to exploit software*** *[Hoglund 04, Koizol 04].*

*This paper discusses the concept of attack patterns as a mechanism to capture and communicate the attacker's perspective. Attack patterns are descriptions of common methods for exploiting software. They derive from the concept of design patterns [Gamma 95] applied in a destructive rather than constructive context and are generated from in-depth analysis of specific real-world exploit examples. Through analysis of observed exploits, the following typical information is captured for each attack pattern:*

- *Pattern name and classification*
- *Attack prerequisites*
- *Description*
- *Related vulnerabilities or weaknesses*
- *Method of attack*
- *Attack Motivation-Consequences*
- *Attacker skill or knowledge required*
- *Resources required*
- *Solutions and Mitigations*
- *Context description*
- *References*

*This information can bring considerable value for software security considerations through all phases of the software development lifecycle (SDLC) and other security-related activities, including:*

- *Requirements gathering*
- *Architecture and design*
- *Implementation and coding*
- *Software testing and quality assurance*
- *Systems operation*
- *Policy and standard generation*

## 1.1 Introduction

This paper introduces the concept, generation, and usage of attack patterns as a valuable knowledge tool in the design, development, and deployment of secure software.

Design patterns are a familiar tool used by the software development community to help solve recurring problems encountered during software development [Gamma 95]. These patterns attempt to address head-on the thorny problems of secure, stable, and effective software architecture and design. Since the introduction of design patterns, many other types of patterns relevant to software have been conceived, including a relatively new construct known as attack patterns [Hoglund 04].

Attack patterns apply the problem-solution paradigm of design patterns in a destructive rather than constructive context. Here, the common problem targeted by the pattern represents the objective of the software attacker, and the pattern's solution represents common methods for performing the attack. Techniques for exploiting software tend to be few and fairly specific [Hoglund 04]. Attack patterns describe the techniques that attackers may use to break software.

The incentive behind using attack patterns is that software developers must think like attackers to anticipate threats and thereby effectively secure their software. Due to the absence of information about software security in many curricula and the traditional shroud of secrecy surrounding exploits, software developers are often ill-informed in the field of software security and especially software exploitation. The concept of attack patterns can be used to teach the software development community how software is exploited in reality and to implement proper ways to avoid the attacks.

Often, security policy also lacks a comprehensive understanding of the issues surrounding software security, as developers have a natural propensity to think in terms of features and functions. Widely accepted and implemented policies that tout encryption as a silver bullet for security problems are an example. Company representatives commonly reassure clients that their data are protected because the database in which it is stored is encrypted. With the hype surrounding firewalls and encryption, it is difficult for the software development community to learn how to actually build secure software. This paper will demonstrate the use of one key tool for effectively building secure software in the absence of any silver bullets.

### Terminology

A lot of terminology used in software security has not been standardized. Different publications use different terminology to describe the same concepts and sometimes even use the same terminology to describe different concepts. Furthermore, marketing literature often misuses security-related terms to sell particular products, adding to the confusion surrounding software security. This section attempts to mitigate the issue for the purposes of this paper. It will briefly describe the essential terminology used, which is mostly borrowed from *Exploiting Software* [Hoglund 04]. The Attack Patterns Glossary should be consulted for a more complete list of terminology used in this paper.

**target software** Target software is software that is the target of an attack.

**target host** A target host is the computer or platform that is running the target software of an attack. A host may be attacked through the interfaces provided by the target software or through purely network-based attack mechanisms.

**exploit** An exploit is a technique or software code (often in the form of scripts) that takes advantage of a vulnerability or security weakness in a piece of target software.

**attack** An attack is the act of carrying out an exploit.

**attacker** An attacker is the person or agent that actually executes an attack. Attackers may range from very unskilled individuals leveraging automated attacks developed by others ("script kiddies") to well-funded government agencies or even organized criminals with extensive software backgrounds.

**attack pattern** An attack pattern is a general framework for carrying out a particular type of attack, such as a method for exploiting a buffer overflow or an interposition attack that leverages certain kinds of architectural weaknesses. In this paper, an attack pattern describes the approach used by attackers to generate an exploit against software.

## Context

Before beginning a discussion on attack patterns, we first need to discuss why attack patterns are important. Attack patterns provide a way for software developers to learn about how their software may be attacked. Armed with knowledge about possible or probable attacks, developers can take steps to mitigate the likelihood or impact of these attacks.

### *Challenges*

Many challenges inhibit the development of secure software. These challenges include

- the actual difficulty of building secure software,
- market forces that favor functionality and time to market over security, and
- a significant knowledge gap between the "black hat"[1]

#### The Attacker's Advantage

The primary challenge in building secure software is that it is much easier to find vulnerabilities in software than it is to make software secure. As an analogy, consider a bank vault. Its designers need to ensure that it is safe against many different types of attacks, not just the seemingly obvious ones. It must generally be safe against mechanical attacks (e.g., using bulldozers), explosives, and safe cracking, to name a few, while still maintaining usability (i.e., allowing authorized personnel to enter, having sufficient ventilation and lighting). This is clearly not a trivial task. However, the attacker may simply need to find one exploitable vulnerability to achieve his or her goal of entering the vault. The attacker may try to access the vault through various potential means, including through the main entrance by cracking the safe combination, through the ceiling, by digging underground, by entering through the ventilation system, by bribing

[1] The term "black hat" invokes the old western movie imagery of the villain in the black cowboy hat and is used to describe individuals who maliciously attack software. attacking community and the defending software development community with a lack of basic awareness of security issues and solutions.

an authorized employee to open the vault, or by creating a small fire in the bank while the vault is open to cause all employees to flee in panic. Given these realities, it is evident that building and maintaining bank vault security is typically much more difficult than breaking into one.

Building secure software has similar issues, but the problem is exacerbated by the virtuality of software. With many systems, the attacker may actually possess the software (obtaining a local copy to attack is often trivial) or could attack it from anywhere in the world through networks. With the ability to attack remotely and without physical access, attacks become much easier. Audit trails may not be sufficient to catch attackers after an attack takes place, because attackers could leverage the anonymity of an unsuspecting user's wireless network or public computers to launch attacks.

Given the greater risks that software faces compared to physical objects, it is essential that software be built with security in mind. To do this, the developers must have a solid understanding of the attacker's perspective to anticipate and thwart expected types of attacks. This is especially true when the assets protected by the software are just as valuable as physical assets protected in bank vaults. Just as bank vaults are built considering all known high-risk attacks that they may face, software should be built considering all applicable known types of attack.

### Functionality Over Security

Another challenge is market forces that demand software developers to maximize functionality and minimize time to market. Functionality is what generally sells software, and security is usually treated as an afterthought. Because users do not see most security capabilities, they are not usually considered a priority.

The most successful products tend to be those that offer the most functionality and enter the market before their competitors'. Unfortunately, this holds true for security products such as encryption software, anti-virus software, firewall software, etc. The products offering the best functionality are often chosen over the ones that offer the best security. Because of this, more and more systems are being exploited with significant newsworthy consequences. As time passes, the shortsightedness of this approach is becoming clear to the industry, but it will still remain a challenge for many years to come.

### The Knowledge Gap

A final central challenge in the area of software security arises from the fact that attackers have been learning how to exploit software for several decades, but the general software development community has not kept up with the knowledge that attackers have gained. This knowledge gap is also evident in the difference of perspective between attackers with their cynical deconstructive view and developers with their happy-go-lucky "you're not supposed to do that" view. The problem continues to grow in part because of the traditional fear that teaching how software is exploited could actually reduce the security of software by helping the existing attackers and even potentially creating new ones. The software development community hoped, in the past, that obscurity would keep the number of attackers relatively small. This assumption has been shown to be a poor one, and some elements of the community are now beginning to look for more effective methods of addressing this problem.

Of course, many other issues pose challenges for software security, but the challenges described here are among the most significant. A basic understanding of the attacker's perspective will help to address these challenges.

### *Solution*

One potential solution to these challenges is using attack patterns to help others understand the attacker's perspective. The black hat community is already well-versed in the techniques used to attack software, but the software development community is not generally educated in the ways in which software is exploited. Attack patterns provide a coherent way of teaching designers and developers how their systems may be attacked and how they can effectively defend them.

A common problem is that software developers try to harden small pieces of software while leaving gaping holes in the big picture. For instance, a developer may use 256-bit AES encryption to secure data but then store the key in the application itself. An attacker will of course choose the easiest way to break software. If an attacker needs the key, he/she will not attempt a brute force attack (computationally infeasible) or cryptanalysis (unlikely to be successful). The attacker will simply obtain the key from the code (very easy).

Likewise, builders of secure physical systems, based on centuries of experience, generally know that attackers always choose the easiest way to achieve their goal. As an analogy, a burglar breaking into a house will not pick the lock(s) on the front door and try to guess the code to the security system if he/she can instead cut the phone line to the house (thus disabling the security system) and break a window to gain access to the inside. Thus, the task of making a house more secure should not involve only better locks and longer security system unlocking codes; they should also involve things like stronger windows and cellular backups for the security system (note that cellular signals also can be jammed, although it is currently not quite as easy as cutting a wire), which can help mitigate known likely attacks. Unless software developers understand similar issues in software security, they cannot effectively build secure software.

Attack patterns help to categorize attacks in a meaningful way, such that problems and solutions can be discussed effectively. Instead of taking an ad hoc approach to software security, attack patterns can identify the types of known attacks to which an application could be exposed so that mitigations can be built into the application.

Another benefit of attack patterns is that they contain sufficient detail about how attacks are carried out to enable developers to help prevent them. Attack patterns, however, do not typically contain inappropriately specific details about the actual exploits to ensure that they do not help educate less skilled members of the black hat community (e.g, script kiddies). Information from attack patterns generally cannot be used directly to create automated exploits.

Of course, attack patterns are not the only useful tool for building secure software. Many other tools, such as misuse/abuse cases, security requirements, threat models, knowledge of common weaknesses and vulnerabilities, coding rules, and attack trees, can help. Attack patterns play a unique role amid this larger architecture of software security knowledge and techniques and will be the focus of this paper.

## Background

### *Origins*

The concept of attack patterns was derived from the notion of design patterns introduced by Christopher Alexander during the 1960s and 1970s and popularized by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in the book *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma 95]. The book discusses vetted solutions to specific problems encountered in object-oriented software design and how to package these solutions for broad leverage in the form of design patterns. A design pattern captures the context and high-level detail of a general repeatable solution to a commonly occurring problem in software design. It is not a low-level design that can be transformed directly into code; it is a description of how to solve a problem that can be used in many situations. Examples of design patterns include the singleton pattern and the iterator pattern. Discussion of these and other specific design patterns is out of scope for this paper but constitutes recommended reading for anyone desiring a full foundational grounding in the context behind attack patterns.

Since the introduction of design patterns, the pattern construct has been applied to many other areas of software development. One of these areas is software security and representation of the attacker's perspective in the form of attack patterns. The term "attack patterns" was coined in discussions among software security thought-leaders starting around 2001, introduced in the paper *Attack Modeling for Information Security and Survivability* [Moore 01] and was brought to the broader industry in greater detail and with a solid set of specific examples by Greg Hoglund and Gary McGraw in 2004 in their book *Exploiting Software: How to Break Code*.

Since the publication of *Exploiting Software*, several individuals and groups throughout the industry have tried to push the concept forward with varying success. These efforts faced challenges like the lack of a common definition and schema for attack patterns, a lack of diversity in the targeted areas of analysis by the various groups involved, and a lack of any independent body to act as the collector and disseminator of common attack pattern catalogues. This paper, as part of the Build Security In effort sponsored by the U.S. Department of Homeland Security, attempts to provide some coherence of definition and structure. Efforts such as the ongoing DHS-sponsored Common Attack Pattern Enumeration and Classification (CAPEC) initiative will collect and make available to the public core sets of attack pattern instances.

### *Concept*

An attack pattern is an abstraction mechanism for describing how a type of observed attack is executed. Following the pattern paradigm, it also provides a description of the context where it is applicable and then, unlike typical patterns, it gives recommended methods of mitigating the attack. In short, an attack pattern is a blueprint for an exploit. We propose that an attack pattern should typically include the following information:

- **Pattern Name and Classification**: A unique, descriptive identifier for the pattern.
- **Attack Prerequisites**: What conditions must exist or what functionality and what characteristics must the target software have, or what behavior must it exhibit, for this attack to succeed?
- **Description**: A description of the attack including the chain of actions taken.

- **Related Vulnerabilities or Weaknesses**: What specific vulnerabilities or weaknesses (see the glossary for definitions) does this attack leverage? Specific vulnerabilities should reference industry-standard identifiers such as Common Vulnerabilities and Exposures (CVE) number, US-CERT number, etc. Specific weaknesses (underlying issues that may cause vulnerabilities) should reference industry-standard identifiers such as the Common Weakness Enumeration (CWE).
- **Method of Attack**: What is the vector of attack used (e.g., malicious data entry, maliciously crafted file, protocol corruption)?
- **Attack Motivation-Consequences**: What is the attacker trying to achieve by using this attack? This is not the end business/mission goal of the attack within the target context but rather the specific technical result desired that could be leveraged to achieve the end business/mission objective. This information is useful for aligning attack patterns to threat models and for determining which attack patterns from the broader set available are relevant for a given context.
- **Attacker Skill or Knowledge Required**: What level of skill or specific knowledge must the attacker have to execute such an attack? This should be communicated on a rough scale (e.g., low, moderate, high) as well as in contextual detail of what type of skills or knowledge are required.
- **Resources Required**: What resources (e.g., CPU cycles, IP addresses, tools, time) are required to execute the attack?
- **Solutions and Mitigations**: What actions or approaches are recommended to mitigate this attack, either through resistance or through resiliency?
- **Context Description**: In what technical contexts (e.g., platform, OS, language, architectural paradigm) is this pattern relevant? This information is useful for selecting a set of attack patterns that are appropriate for a given context.
- **References**: What further sources of information are available to describe this attack?

Two examples of attack patterns are provided below [Hoglund 04]:

1. **Pattern name and classification**: Make the Client Invisible
   - **Attack Prerequisites**: The application must have a multi-tiered architecture with a division between client and server.
   - **Description**: This attack pattern exploits client-side trust issues that are apparent in the software architecture. The attacker removes the client from the communication loop by communicating directly with the server. This could be done by bypassing the client or by creating a malicious impersonation of the client.
   - **Related Vulnerabilities or Weaknesses**: CWE–Man-in-the-Middle (MITM), CWE- Origin Validation Error, CWE- Authentication Bypass by Spoofing, CWE- No Authentication for Critical Function, CWE- Reflection Attack in an Authentication Protocol
   - **Method of Attack**: Direct protocol communication with the server.
   - **Attack Motivation-Consequences**: Potentially information leak, data modification, arbitrary code execution, etc. These can all be achieved by bypassing authentication and filtering accomplished with this attack pattern.

- **Attacker Skill or Knowledge Required**: Finding and initially executing this attack requires a moderate skill level and knowledge of the client-server communications protocol. Once the vulnerability is found, the attack can be easily automated for execution by far less skilled attackers. Skill level for leveraging follow-on attacks can vary widely depending on the nature of the attack.
- **Resources Required**: None, although protocol analysis tools and client impersonation tools such as netcat can greatly increase the ease and effectiveness of the attack.
- **Solutions and Mitigations**:
- Increase Resistance to Attack: Utilize strong two-way authentication for all communication between client and server. This option could have significant performance implications.
- Increase Resilience to Attack: Minimize the amount of logic and filtering present on the client; place it on the server instead. Use white lists on server to filter and validate client input.
- **Context Description**: "Any raw data that exist outside the server software cannot and should not be trusted. Client-side security is an oxymoron. Simply put, all clients will be hacked. Of course the real problem is one of client-side trust. Accepting anything blindly from the client and trusting it through and through is a bad idea, and yet this is often the case in server-side design" [Hoglund 04].
- **References**: *Exploiting Software: How to Break Code*, p.150 [Hoglund 04].

2. **Pattern name and classification**: Shell Command Injection—Command Delimiters

- **Attack Prerequisites**: The application must pass user input directly into a shell command.
- **Description**: Using the semicolon or other off-nominal characters, multiple commands can be strung together. Unsuspecting target programs will execute all the commands. An example may be when authenticating a user using a web form, where the username is passed directly to the shell as in:

-
  ```
  exec( "cat data_log_" + userInput + ".dat")
  ```
- The "+" sign denotes concatenation. The developer expects that the user will only provide a username. However, a malicious user could supply "username.dat; rm –rf / ;" as the input to execute the malicious commands on the machine running the target software. Similar techniques are also used for other attacks such as SQL injection. In the above case, the actual commands passed to the shell will be:
  ```
  cat data_log_username.dat; rm –rf /; .dat
  ```
- The first command may or may not succeed; the second command will delete everything on the file system to which the application has access, and success/failure of the last command is irrelevant.
- **Related Vulnerabilities or Weaknesses**: CWE-OS Command Injection, CVE-1999-0043, CVE-1999-0067, CVE-1999-0097, CVE-1999-0152, CVE-1999-0210, CVE-1999-0260, 1999-0262, CVE-1999-0279, CVE-1999-0365, etc.
- **Method of Attack**: By injecting other shell commands into other data that are passed directly into a shell command.

  - **Attack Motivation-Consequences**: Execution of arbitrary code. The attacker wants to use the target software, which has more privilege than the attacker, to execute some commands that he/she does not have privileges to execute.
  - **Attacker Skill or Knowledge Required**: Finding and exploiting this vulnerability does not require much skill. A novice with some knowledge of shell commands and delimiters can perform a very destructive attack. A skilled attacker, however, may be required to subvert simple countermeasures such as rudimentary input filtering.
  - **Resources Required**: No special or extensive resources are required for this attack.
  - **Solutions and Mitigations**: Define valid inputs to all fields and ensure that the user input is always valid. Also perform white-list and/or black-list filtering as a backup to filter out known command delimiters.
  - **Context Description**: OS: UNIX.
  - **References**: *Exploiting Software* [Hoglund 04].

Note that an attack pattern is not overly generic or theoretical. The following is not an attack pattern: "writing outside array boundaries in an application can allow an attacker to execute arbitrary code on the computer running the target software." The statement does not identify what type of functionality and specific weakness is targeted or how malicious input is provided to the application. Without that information, the statement is not particularly useful and cannot be considered an attack pattern.

An attack pattern is also not an overly specific attack that only applies to a particular application. For instance, "When the PATH environment variable is set to a string of length greater than 128, the application foo executes the code at the memory location pointed to by characters 132, 133, 134, and 135 in the environment variable." This amount of specificity is dangerous to disclose and provides limited benefit to the software development community. It is dangerous because it enables black hats to more easily attack particular software without requiring much thought. It is of limited benefit to the software development community because it does not help them discover and fix vulnerabilities in other applications or even fix other similar vulnerabilities in the same application.

Though not broadly required or typical, it can be valuable to adorn attack patterns where possible and appropriate with other useful reference information such as:

- **Examples-Instances**: Explanatory examples or demonstrative exploit instances of this type of attack. They are intended to help the reader understand the nature, context and variability of the attack in more practical and concrete terms.
- **Source Exploits**: From which specific exploits (e.g., malware, cracks) was this pattern derived and which shows an example?
- **Related Attack Patterns**: What other attack patterns affect or are affected by this pattern?
- **Relevant Design Patterns**: What specific design patterns are recommended as providing resistance or resilience to this attack, or which design patterns are not recommended as they are particularly susceptible to this attack?
- **Relevant Security Patterns**: What specific security patterns are recommended to provide resistance or resilience to this attack?

- **Related Guidelines or Rules**: What existing security guidelines or secure coding rules are relevant to identifying or mitigating this attack?
- **Relevant Security Requirements**: Have specific security requirements relevant to this attack been identified which offer opportunities for reuse?
- **Probing Techniques**: What techniques are typically used to probe and reconnoiter a potential target to determine vulnerability and/or to prepare for an attack?
- **Indicators-Warnings of Attack**: What activities, events, conditions, or behaviors could serve as indicators that an attack of this type is imminent, in progress, or has occurred?
- **Obfuscation Techniques**: What techniques are typically used to disguise the fact that an attack of this type is imminent, in progress, or has occurred?
- **Injection Vector**: What is the mechanism and format for this input-driven attack? Injection vectors must take into account the grammar of an attack, the syntax accepted by the system, the position of various fields, and the acceptable ranges of data [Hoglund 04].
- **Payload**: What is the code, configuration, or other data to be executed or otherwise activated as part of this injection-based attack?
- **Activation Zone**: What is the area within the target software that is capable of executing or otherwise activating the payload of this injection-based attack? The activation zone is where the intent of the attacker is put into action. The activation zone may be a command interpreter, some active machine code in a buffer, a client browser, a system API call, etc. [Hoglund 04].
- **Payload Activation Impact**: What is the typical impact of the attack payload activation for this injection-based attack on the confidentiality, integrity, or availability of the target software?

### Related Concepts

There exist many other concepts and tools related to attack patterns, including fault trees, attack trees, threat trees, and security patterns that are available to the community. It is useful to examine and describe these concepts briefly to reduce confusion between these concepts and attack patterns and so that related literature can be used as a reference when researching or using attack patterns.

Bell Labs developed the concept of fault trees for the Air Force in 1962. It was later applied in a software context in the works of Nancy Leveson [Leveson 83] in the early 1980s. Fault trees provide a formal and methodical way of describing the safety of systems, based on various factors affecting potential system failure. Fault trees are commonly used in safety engineering; the goal of which is to ensure that life-critical systems behave as required when parts of them fail [Vesely 81]. Fault trees have system failure as their root node and potential causes of system failure as other nodes in the tree. Any particular node's "children" represent ways in which the node can "fail." The concept of fault trees is especially helpful for analyzing software for which availability/survivability is a major security concern. Fault trees are a fairly mature concept, and an abundance of literature elaborates on the topic. Fault trees and attack patterns have only a very tenuous relationship. Attack patterns are much more closely aligned with attack trees, a derivative of fault trees, which are described below.

The concept of attack trees was first promulgated by Bruce Schneier, CTO of Counterpane Internet Security. Attack trees are similar to fault trees, except that attack trees are used to analyze the security of systems rather than safety. Attack trees provide a formal and methodical way of describing the security of systems based on varying attacks [Schneier 99]. Microsoft uses the term "threat tree" to describe the same concept [Swiderski 04]. An attack tree has the attacker's goal as the root, and the children of each parent node represent conditions of which one or more must be satisfied to achieve the goal of the parent node. In this manner, all paths to the root from the leaf nodes indicate potential attacks.

An attack pattern consists of a minimal set of nodes in an attack tree that achieves the goal at the root node. In a tree with only "or" branches, this consists of all paths from a leaf node to the root node. Such paths are also known as "attack paths." In a tree with some "and" branches, an attack pattern may be a subtree of the attack tree that includes the root node and at least one leaf node.

Attack trees and attack patterns are complementary concepts that balance and enhance each other. While attack trees provide a holistic view of the potential attacks facing a particular piece of software, attack patterns provide actionable detail on specific types of common attacks potentially affecting entire classes of software. Details and examples of attack trees can be found in [Schneier 99].

Lastly, another concept related to attack patterns is security patterns. Security patterns consist of general solutions to recurring security problems. A security pattern encapsulates security expertise in the form of vetted solutions to these recurring problems, presenting issues and tradeoffs in the usage of the pattern [Kienzle 01]. Examples include implementing account lockout to prevent brute force attacks, secure client data storage, and password authentication. Because general software developers may not be familiar with security best practices or with security issues, security patterns attempt to provide practical solutions that can be implemented in a straightforward manner. Security patterns also list various tradeoffs in the solutions. Security patterns can be an effective complement to attack patterns in providing viable solutions to specific attack patterns at the design level. As such, it should be noted that security patterns generally describe relatively high-level repeatable implementation tasks such as user authentication and data storage. They are not typically suitable for low-level implementation details such as NULL termination of strings or even very high-level design issues such as client-side trust issues. Hence, they are excellent for describing solutions to programming problems with a security context but they do not demonstrate how to avoid most common software development pitfalls. A security patterns repository is available at SecurityPatterns.org. The repository is not meant to be a comprehensive or most up-to-date list of security patterns.

## 1.2 Attack Pattern Generation

The Introduction section presented introductory and contextual information on attack patterns. This section describes a typical process for how attack patterns are actually generated. The intended audience for this section includes mainly security researchers and experienced security practitioners who are interested in discovering and documenting new attack patterns. It is, however, also valuable for the broader audience in that it gives a much deeper understanding of the source and meaning of attack patterns.

### Purpose

As attackers become more sophisticated, they will discover new ways of exploiting software. In addition, new software and development environments will introduce new types of vulnerabilities that presently may be unknown. To ensure that the software development community continues to implement effective countermeasures against the latest known attacks, it is important to analyze the latest exploits to see whether they represent any new types of attacks. Only after the attacks are characterized can effective countermeasures against those types of attacks be designed and implemented.

In addition, analyzing the latest exploits and generating new attack patterns is an essential prerequisite to the creation of effective security policies. Policy developers should be aware of all known attacks that a system may face before they attempt to develop relevant and complete security policies.

While attack patterns fundamentally represent common approaches to exploits, they do not necessarily need to be generated only from actual exploits discovered in the wild. In cases where organizations are performing security-related research, they may discover a new way to attack a system. This knowledge can be used internally by the organization (or provided to vendors) to mitigate the issues before attackers discover them. The example presented at the end of this section was actually discovered in this manner.

### Inputs

The process of attack pattern generation begins when a new exploit is discovered that does not match one of the known attack patterns. The inputs to the process include the actual exploits (if available), any existing vendor patches for the exploits, forensic information, and the existing knowledge base of attack patterns. The exploits may be discovered in the wild, or they may be generated by security researchers in an attempt to discover exploits before attackers do.

### Analysis

The exploit analysis process typically consists of the following steps:

1. Analyze the exploit through reverse engineering, forensic analysis, and analyzing any available patches by vendors of the target software. This step is not specific to attack pattern analysis and is generally performed to understand exploits and develop countermeasures such as antivirus definitions and spyware removal tools. Once the inner workings of the exploit are revealed, actual attack pattern analysis can begin.
2. Determine whether the exploit is an instantiation of any existing attack patterns. This is often not a clear and unambiguous decision. A careful analysis and comparison must be performed. In most cases where the exploit is discovered in the wild, it will be an existing attack pattern and the analysis will stop here. Otherwise, a new attack pattern will have been discovered, which should be analyzed and documented as described below.
3. Determine the functionality in the software that contained the vulnerability. The functionality could be a file parser, format converter, cookie handler, or anything else. Determine whether the exploit attacks a vulnerability or weakness in the particular functionality or whether the same issue could exist in the software even if the targeted functional component were removed. If the exploit targets specific functionality, then attempt to generalize the attack. For example, if an attack targets an MP3 player,

then could similar attacks also be used against WMV players, JPEG viewers, ZIP file extractors, etc? The vulnerability could lie in *any* binary file processor, or it could exist *only* in MP3 playing software.

4. Determine how the software vulnerability was exploited. Examples include providing a maliciously crafted file to the software, leveraging a race condition, providing separator characters in the input, or bypassing client-side input filtering. This step helps to identify how the targeted functionality determined in the previous step was exploited.
5. Determine what skill level or knowledge the attacker would need to execute such an attack. Note that there may be different skill levels and knowledge required to generate certain results. For example, exploiting a buffer overflow to crash a system may require very little skill, but actually executing malicious code on the target host to gain control of it may require a highly skilled attacker.
6. Determine the resources required to execute the attack. Does the attack simply require an attacker manually entering commands at a terminal, or does it involve compromising thousands of hosts before using them to attack the main target? Would execution of the attack require a well-funded organization's support? It is important to determine the resources required to execute an attack, as it helps determine the likelihood of an attack and helps to prioritize mitigations during actual software development.
7. Determine the motivation of the attacker that generates this type of exploit. Why would an attacker choose this type of attack in particular? Given a choice between various technical (e.g., executing a buffer overflow) and non-technical (e.g., social engineering) means of achieving a goal, attackers tend to select the easiest ones. Keeping that in mind, determine what makes the particular attack attractive. What does the attacker have to begin with, and what does the attacker want to accomplish? This discussion should be mostly technical, because business consequences will obviously depend on the particular software and a deployed environment. The consequences may include execution of arbitrary code on target host, denial of service, obtaining privileged access to target host, etc.

### Evaluation

Once a new attack pattern is generated, it should be evaluated to ensure that it models the applicable exploits well. It is essential that the evaluation be performed by a different person than the one who generated the attack pattern. Otherwise, the attack pattern author could potentially gloss over issues due to implicit assumptions made during the analysis. The first step is to ensure that a pre-existing attack pattern does not model the exploits. If a pre-existing attack pattern that models the exploit is found, determine why the new attack pattern was created and whether it would be more beneficial to amend the existing attack pattern than to create a new one.

Assuming that the attack pattern is new, ensure that the exploits from which it was generated are actually instantiations of the attack pattern. If not, determine what modifications to the attack pattern are required to correct the problem. If this is done, then examine the unmodified attack pattern as well to determine whether it is a valid attack pattern that may be applicable in other scenarios.

The next step is to ensure that the attack pattern is not overly generic. Questions that may help to determine this include:

- Does the attack pattern describe what part of the software is attacked?
- Does the attack pattern describe how malicious input is provided to the target software?
- Will knowledge of the attack pattern help a software designer or developer avoid the problem?

If the answer to any of the above questions is “no,” then the attack pattern is likely too generic. For instance, an overly generic attack pattern may state “providing an unexpectedly large input to the application causes it to crash.” This “attack pattern” does not describe what part of the software is attacked or how malicious input is provided to the target software. Developers do not learn what input they need to validate (e.g., input from text fields in a Windows GUI application, input from a web form, input from a binary resource file). It provides no indication as to what particular source of input is untrusted, and hence must be validated. A potential attacker also would be unable to use this attack pattern because it tells them absolutely nothing about how malicious input could be provided to the application. This may be considered positive, but the fact is that many attackers are likely already familiar with the attack, along with specific exploit instances. At the same time, if the attack pattern is completely useless to an unskilled attacker, it likely does not represent a valuable capture of the attacker’s perspective. More details can be provided to make the attack pattern helpful to software developers, while keeping particular attack details private.

The next step is to ensure that the attack pattern is not overly specific. Questions that may help to determine this include:

- Can the attack pattern be used to find previously undiscovered vulnerabilities in software?
- Can the attack pattern be used by relatively unskilled attackers to exploit software?

If the answer to the first question is “no,” then the attack pattern is likely too specific. If the answer to the second question is “yes,” then either the attack is extremely simple (such as a command line delimiter attack) or the attack pattern is too specific and detailed. If the attack pattern is found to be too specific or detailed, this problem should be mitigated. An overly specific attack pattern is likely to benefit only attackers and provide little ongoing value to developers.

The accessibility of the attack pattern also should be evaluated; one of the target audiences is software designers and developers that may have little or no training in security issues. It would be easy for security researchers to develop attack patterns that are completely inaccessible to designers and developers. It is important to keep the goal of creating attack patterns in mind: attack patterns are designed to help software designers and developers with little security experience to understand security issues so that they can develop secure software.

### Outputs

The typical schematic structure and content for an attack pattern that comes out of this process is described in the Introduction section above. In addition to the information described there, the attack pattern should clearly indicate the author(s) and reviewer(s) of the attack pattern. Depending on the environment in which the attack pattern was developed, it should be published either to a private company repository or to a public attack patterns repository. This will ensure that the attack pattern will be easily accessible and will not get lost.

### Example

We will illustrate the attack pattern generation process using an example. The example used here was chosen for its simplicity of explanation and understanding and was, in reality, discovered through other means.

Suppose that there have recently been many reports of *gzip*-compressed files that have been causing most virus scanners to crash. In addition, receiving *gzip*-compressed HTML data is causing popular web browsers to crash. A security researcher, Alice, has been given the task of investigating the issues and determining whether they are related.

Alice is already familiar with existing attack patterns that are public knowledge (assume that this particular attack is not public knowledge). She begins by obtaining some problematic *gzip*-compressed files. She tries to decompress one and notes that the decompression utility takes an unusually long time. She obtains a file listing of the directory in which the file is being decompressed and notes that the file being output by the decompression utility is currently over 1 gigabyte in size. Realizing that this is unusual, she stops the decompression utility and opens the partially decompressed file in a hex editor. The decompressed file seems only to contain the letter ”A,” repeated over and over. Knowing how run-length encoding works, Alice realizes that the decompressed file must simply contain the same byte repeated billions of times, so that the compression is extremely efficient. The compressed file more or less need only contain the letter ”A,” along with the number of times it is repeated in the original file. Thus, the compressed file can be extremely small (several kilobytes to several megabytes), but the decompressed file can be several hundred gigabytes in size. When a virus scanner attempts to scan the contents of the compressed file, it must decompress it in memory first. Most computers do not have several hundred gigabytes of memory and eventually run out of memory and crash. Alice discovers the same problem at malicious websites that send a small amount of *gzip*-encoded HTML data to the clients, causing the browsers that attempt to decompress and display the data to crash.

Now that Alice has discovered the cause of the attack and knows that the attack pattern does not currently exist, she decides to generate an attack pattern that describes the attack. She determines that the functionality under attack is decompression of *gzip*-encoded data. However, she also realizes that *gzip* is not the only type of encoding that may be susceptible to such attacks. Other encoding formats, including *ZIP*, *bzip2*, *PNG*, and *GIF*, also use run-length encoding to compress data and could be vulnerable to such attacks. In fact, any type of compression where the efficiency of compression is not bounded by a reasonable value may be vulnerable to such attacks. Thus, Alice determines that any software that performs decompression may be vulnerable to such attacks.

Next, Alice must determine how the vulnerability is exploited. She determines that this problem could occur any time a target host attempts to decompress malicious data. All an attacker must do is activate software on the target host to attempt to decompress the data in memory. Software that may do this includes virus scanners, image viewers, and web browsers. Thus, the attacker must somehow place the malicious data on the target host. In most cases, a utility such as a virus scanner would automatically do the rest. In fact, this can be used to cause a fairly large-scale denial-of-service attack. The malicious file can be e-mailed to an address in a target corporation, and the virus scanner on the e-mail server will cause the server to crash. Even if backup servers exist, they will also crash when attempting to pick up where the other server left off (i.e., when they attempt to scan the malicious file). This will effectively cause a denial of service and will halt all e-mail communication within that organization.This issue is now public knowledge and has been mitigated in all popular antivirus software. Public release of this information in this paper no longer poses a significant threat.

Now, Alice must determine the skill level of the attacker that may be able to execute such an attack. She notes that crafting a malicious file is not an simple task, as the attacker cannot just create a file that is several hundred gigabytes in size and then compress it (due to memory limitations of the attacker's computer). The attacker must know details of the compressed file format so that he/she can craft a valid but malicious compressed file without having to create the original decompressed file. However, if such a malicious file is ever obtained by an unskilled attacker, he/she can easily leverage it to attack other systems. Thus, creating a malicious file requires a skilled attacker, but even an unskilled attacker can use the file to execute an attack.

Alice next must determine the resources required to execute the attack. This particular attack requires minimal resources--simply that the attacker be able to send data to the target, which could be done via e-mail, HTTP, FTP, etc.

Now that Alice knows the attack details, she needs to determine why an attacker would execute such an attack. Identifying the motivation helps to determine the relevance of this attack pattern to future situations. This attack is focused on achieving a denial of service. An attacker wanting to disrupt all e-mail communication in an organization, either for simple mischief or to cause losses, may try to execute this attack. The attack is extremely simple, and a single e-mail can halt all e-mail communication within an organization regardless of the number of backup mail servers. The attacker may also want to deny all users access to a message board by posting a malicious picture on the message board that causes users' browsers to crash when they attempt to view it. A professional attacker wishing to monetize such an attack could even leverage this attack in combination with other business actions, such as bringing down a T-bill auction at an opportune time to manipulate monetary results. There are several such attacks where an attacker can deny the target access to some resource for a limited period of time.

The following describes the attack pattern:

1. **Pattern name and classification**: Denial of Service – Decompression Bomb
   - **Attack Prerequisites**: The application must decompress compressed data. For the attack to be maximally effective, the decompression should happen automatically without any user intervention.
   - **Description**: The attacker generates a small amount of compressed data that will decompress to an extremely large amount of data. The compressed data may only be a few kilobytes in size, whereas the decompressed data may be several hundred gigabytes in size. The target is running software that automatically attempts to decompress the data in memory to analyze it (such as with antivirus software) or to display it (such as with web browsers). When the target software attempts to decompress the malicious data in memory, it runs out of memory and causes the target software and/or target host to crash.
   - **Related Vulnerabilities or Weaknesses**: CWE-Data Amplification, CVE-2005-1260

- **Method of Attack**: By maliciously crafting compressed data and sending it to the target over any protocol (e.g., e-mail, HTTP, FTP).
- **Attack Motivation-Consequences**: The attacker wants to deny the target access to certain resources.
- **Attacker Skill or Knowledge Required**: Creating the exploit requires a considerable amount of skill. However, once such a file is available, an unskilled attacker can find vulnerable software and attack it.
- **Resources Required**: No special or extensive resources are required for this attack.
- **Solutions and Mitigations:** Restrict the size of output files when decompressing to a reasonable value. Especially, handle decompression of files with a large compression ratio with care. Builders of decompressors could specify a maximum size for decompressed content and then cease decompression and throw an exception if this limit is ever reached.
- **Context Description:** Any application that performs decompression of compressed data in any format (e.g., image, archive, sound, gzip-ed HTML)
- **References:** Decompression bomb vulnerabilities

## 1.3 Attack Pattern Usage

Unlike many other concepts and tools with a narrowly focused area of impact, attack patterns provide potential value during all phases of software development regardless of the SDLC chosen, including requirements, architecture, design, coding, testing, and even deploying the system. However, because attack patterns describe how an attacker may break software, some readers may not immediately understand how attack patterns can be used to actually build secure software. Once the reader grasps the importance of understanding the attacker’s perspective to software security, the value of attack patterns becomes intuitively clear. Without knowing how software may be attacked, it is difficult to know how to defend against the attacks.

The sections below describe how attack patterns can be leveraged during each stage of the SDLC. To make the information more concrete, each section provides an example. All examples will use as their basis one application that needs to be developed. The application will be web based and designed to let consumers purchase books online.

### Requirement Gathering

This paper assumes that the reader is familiar with the basic activities and results of typical software requirements definition efforts. Many other resources explain the various methodologies and challenges for requirements gathering. The Build Security In site offers a good Requirements Engineering Annotated Bibliography. Discussion here will focus on the role attack patterns play in defining more appropriate and comprehensive requirements regarding the security of the software under development.

#### *Functional Requirements*

Most requirements gathering starts with relatively high-level functional requirements such as “users shall be able to access the site using at least the latest versions of Internet Explorer and Mozilla Firefox” and “users shall be able to purchase books in any currency”. These high-level requirements generally lead to more detailed functional requirements and can potentially drive out security requirements. These security requirements can be functional, whether visible to the end user or not, or not functional in nature, but equally important. Very often, detailed functional and non-functional requirements including security requirements are overlooked and neglected because the general focus is basic functionality.

### Deriving Security Requirements From Functional Requirements

The above two requirements should lead to questions that could help identify security requirements. If a user attempts to view the website with anything but the latest versions of Internet Explorer and Mozilla Firefox, what should happen? Is it acceptable if the browser crashes? Is it acceptable if absolutely nothing is displayed? Is there anything that the server needs to do to differentiate between browsers? What should happen if the self-identification data sent by the client is spoofed (e.g., if Mozilla Firefox is set to report itself as being Internet Explorer)? Also, if users can purchase books in other currencies, then should they be able to browse the website in other languages or encoding schemes (e.g., Unicode)? If so, how many languages and encoding schemes should the website support? What should happen if a client sends characters from a language or encoding scheme that the server does not accept?

As shown above, the process of making functional requirements more specific often is also an effective mechanism for identifying security requirements. For instance, indicating that “if a client sends characters from a language that the server does not recognize, then the server will return a HTTP 415 status code” is a good security requirement. This informs the developers how to handle the issue. Otherwise, the problem may be overlooked, causing issues such as attackers being able to bypass input filters.

### Positive and Negative Security Requirements: The Role of Attack Patterns

Security-focused requirements are typically further split between positive requirements, which specify functional behaviors the software must exhibit (often security features), and negative requirements (typically in the form of misuse/abuse cases), which describe behaviors that the software must not exhibit to be operating securely [McGraw 06].

Attack patterns can be an invaluable resource for helping to identify both positive and negative security requirements. They have obvious direct benefit in defining the software’s expected reaction to the attacks they describe. When put into the context of the other functional requirements for the software and when considering the underlying weaknesses targeted by the attack, they can help identify both negative requirements describing potential undesired behaviors and positive functional requirements for avoiding, or at least mitigating, the potential attack. For instance, if a customer provides the requirement “the application must accept ASCII characters,” then the attack pattern “Unicode Encoding” can be used to ask the question “What should the application do if Unicode characters or another unacceptable character set is encountered?” From this question, misuse/abuse cases can be defined such as “Malicious user provides Unicode characters to the data entry field.” By having a specific definition for this negative requirement, the designers, implementers, and testers will have a clear idea of the type of hostile environment with which the software must deal and will build the software accordingly. This information can also help define positive requirements such as “The system shall filter all input for Unicode characters.” If these sorts of requirements are overlooked, the developed application may have instances in which it may unknowingly accept Unicode characters, and an attacker could use that fact to bypass input filters for ASCII characters.

Many vulnerabilities result from vague specifications and requirements. This includes ambiguities outside the immediate scope of the application, including "unspecified behavior" in certain specifications (e.g., C language and how compilers must deal with certain situations) or RFCs (e.g., IP fragmentation and how end nodes interpret the specification in varying fashions). Requirements should specifically address these ambiguities to avoid opening up multiple security holes. In general, attack patterns allow the requirements gatherer to ask “what if” questions to make the requirements more specific. If an attack pattern states “Condition X can be leveraged by an attacker to cause Y,” then a valid question may be “What should the application do if it encounters condition X?”

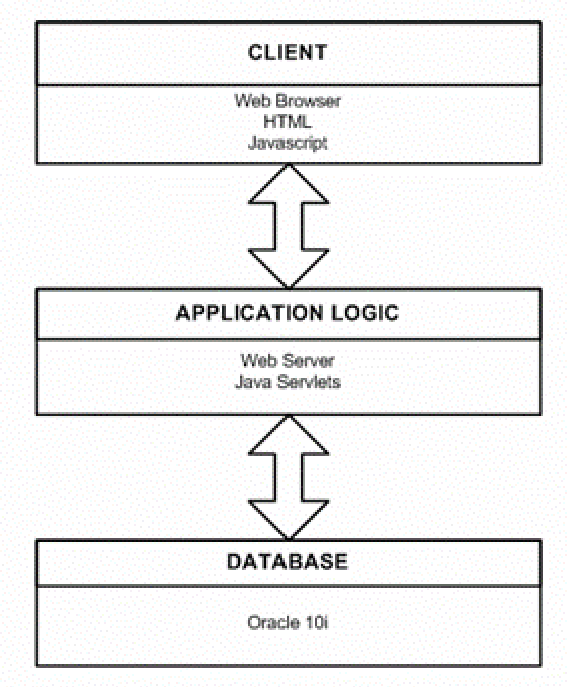
#### Varying Levels of Attack Pattern Detail and Specificity

Attack patterns can exist at varying levels of detail and specificity; they often may start out more abstract with less known instances of exploit and then mature in level of detail over time as more exploit instances are discovered. These differing levels of detail also can influence the requirements they identify at different levels. More abstract attack patterns typically lead to less specific nonfunctional requirements, while more detailed attack patterns typically lead to more specific functional requirements.

### Architecture and Design

Once requirements have been defined, all software must go through some level of architecture and design. Regardless of the formality of the process followed, the results of this activity will form the foundation for the software and drive all remaining development activities. During architecture and design, decisions must be made about how the software will be structured, how the various components will integrate and interact, which technologies will be leveraged, and how the requirements defining how the software will function will be interpreted. Careful consideration is necessary during this activity, as up to 50% of software defects leading to security problems are design flaws [McGraw 06]. In the example in Figure 1, a potential architecture could consist of a three-tier system with the client (a web browser leveraging Javascript/HTML)), a web server (leveraging Java$^{TM}$ Servlets), and a database server (leveraging Oracle 10i). Decisions made at this level can have a significant impact on the overall security profile of the software.

*Figure 0-1: Example architecture*



Attack patterns can be valuable during architecture and design in two ways. First, some attack patterns describe attacks that directly exploit architecture and design flaws in software. For instance, the “Make the Client Invisible” attack pattern described in the Introduction section above exploits client-side trust issues that are apparent in the software architecture. Second, attack patterns at all levels can provide a useful context for the threats that the software is likely to face and thereby determine which architectural and design features to avoid or to specifically incorporate. The Make the Client Invisible attack pattern tells us that absolutely nothing sent back by the client can be trusted, regardless of what network security mechanisms (e.g., SSL) are used. The client is untrusted, and an attacker can send back literally any information that he/she desires. All input validation, authorization checks, etc. must be performed on the server side. In addition, any data sent to the client should be considered visible by the client regardless of its intended presentation (i.e., data that the client should not see should never be sent to the client). Performing authorization checks on the client side to determine what data to display is unacceptable.

The Make the Client Invisible attack pattern instructs the architects and designers to ensure that absolutely no business logic is performed on the client side. In fact, depending on the system requirements, and the threats and risks the system faces, the architects and designers may even want to define an input validator through which all input to the server must pass before being sent to the other classes. Such decisions must be made at the architecture and design phase, and attack patterns provide some guidance regarding what issues should be considered.

It is essential to document any attack patterns used in the architecture/design phase so that the application can be tested using those attack patterns. Tests must be created in the later testing phase to validate that mitigations for the attack patterns considered during this phase were implemented properly.

### Implementation and Coding

If architecture and design have been performed properly, each developer implementing the design should be writing well-defined components with well-defined interfaces.

Attack patterns can be useful during implementation because they identify the specific weaknesses targeted by relevant attacks and allow the developer to ensure that these weaknesses do not occur in their code. These weaknesses could take the form of implementation bugs or simply valid coding constructs that bear with them security implications. Implementation bugs are not always easy to avoid or to catch and fix. Even after applying basic review techniques, they can still remain abundant and can make software vulnerable to extremely dangerous exploits. It is important to extend basic review techniques with more focused security relevant concerns. Failure to properly check an array bound, for example, can lead to an attacker being able to execute arbitrary code on the target host. Failure to perform proper input validation can lead to an attacker being able to destroy an entire database. Underlying security issues in non-buggy valid code are typically more difficult to identify. They cannot be tested for with a functional behavioral model the way bugs can. They require specialized knowledge of what these weaknesses look like. This paper focuses on how attack patterns can be used to identify specific weaknesses for targeting and mitigation through informing the developer ahead of time of the issues to avoid and through providing a list of issues (Security Coding Rules) to look for in code reviews, often performed with security scanning tools.

Prevention requires that the developers understand applicable attack patterns and ensure that their code does not allow the attack patterns to succeed. The first step is to determine which attack patterns are applicable for the application being developed. Only a subset of attack patterns will be applicable for a particular piece of software, depending on its architecture, environment, and the technologies used to implement it. For instance, buffer overflow vulnerabilities are not typically applicable if all coding is done in Java. Input validation vulnerabilities may be less of a concern if all untrusted input is passed through a vetted, central, server-side filter before it is delivered to their code, rather than relying on all entry points (often implemented by different individuals) to perform their own validation. It is important to determine the attack patterns that will be applicable for a particular project. In some instances, different attack patterns may be applicable for different components of a product.

Once the applicable attack patterns are determined, they can be used to guide developers as to what not to allow in their code. In our example, a developer could leverage an attack pattern such as “simple script injection” and avoid XSS vulnerabilities. One relatively easy way to do this is to identify all places from which output is being sent to the user from an untrusted source and convert potentially dangerous characters into their HTML equivalents. For instance, convert ”<” to ”&lt;”, ”>” to ”&gt;”, etc. Third-party libraries for Java can perform such conversions automatically. JavaScript’s escape() function performs a similar task. This will prevent untrusted input containing potentially malicious data from being displayed to the user. Malicious data could include artifacts such as <script> tags inserted by an attacker. This conversion should be carefully managed to avoid potential unintended buffer overflow issues. Of course, this problem could also be handled in other ways, such as use of a white list or at an architectural level by defining an input validator and an output sanitizer. The architectural approach would be more suitable for large projects, whereas dealing with the problem at the implementation level may be acceptable for smaller projects.

Good architecture/design as well as developer awareness, enhanced with attack patterns, can potentially help to minimize many security weaknesses. However, it is also essential to ensure that all source code, once written, is reviewed to validate the absence of targeted weaknesses. Due to the size and monotony of this task, it is typically performed using an automated analysis tool (e.g., those from Fortify, Klocwork, Coverity). Even though analysis tools cannot find all security weaknesses, they can help weed out many potential issues. Using attack patterns as guidance, specific subsets of the tools' search rules can be targeted and custom rules can be created for organizations to help find security weaknesses or instances of failure to follow security standards. For example, revisiting the potential "Simple Script Injection" attack pattern, an organization may have a security standard in which all untrusted input is passed through an input filter, and all output of data obtained from an untrusted source is passed through an encoder. An organization can develop such filters and encoders, and static source code analysis tools can help find occurrences in code where developers may have neglected to adhere to standards and opted to use Java's input/output features directly.

### Software Testing and Quality Assurance

Testing and quality assurance is a critical phase in the software development lifecycle. Software must undergo several levels and types of testing before it is released into a production environment. Different levels of testing include unit testing, integration testing, system testing, regression testing, and deployment testing. Different types of testing include functional testing, security testing (including penetration testing), performance testing, data integrity testing, and stress testing. A detailed discussion about all of the various levels and types of testing is out of scope for this paper. However, it is important to note that attack patterns can be leveraged during many different levels and types of testing to help design test cases.

The testing phase is different than the previous ones in the SDLC in that its goal is not necessarily constructive; the goal of risk-based security testing is typically to attempt to break software so that the discovered issues can be fixed before an attacker can find them [Whittaker 03]. The purpose of using attack patterns in this phase is to have the individuals performing the various levels and types of testing act as attackers attempting to break the software.

#### *Leveraging Attack Patterns in Unit Testing*

Unit testing involves testing the components or pieces of software independently to ensure that they meet their functional and non-functional specifications. Applicable attack patterns should be used to identify relevant targeted weaknesses and to generate test cases for each component to ensure that they avoid or resist these weaknesses. For example, to test for shell command injection using command delimiters, malicious input strings containing delimiter separated shell commands should be crafted and input to the applicable component(s) to ensure proper behavior when provided with this type of malicious data.

#### *Leveraging Attack Patterns in Integration Testing*

Integration testing involves ensuring that software components integrate and interact together properly. This requires not only ensuring that all components compile together and that their interfaces match but also that the actual functionality of the components does not conflict If a good architecture and design are created and proper unit testing is performed, integration testing should reveal less major issues than otherwise. A primary security issue to consider during integration testing is whether the individual components make differing assumptions based on security such that the integrated whole may contain conflicts or ambiguities. Attack patterns can be leveraged to create some test cases for integration testing. At a minimum, the attack patterns documented in the architecture/design phase should be used to create integration tests. Other attack patterns may be applicable as well. For instance, the Make the Client Invisible attack pattern can be used to create test cases that simulate an attacker bypassing the client and communicating directly with the server or an attacker modifying the client to send malicious data to the server.

### *Leveraging Attack Patterns in System Testing*

System testing is used to test the entire system to ensure that it meets all of its functional and non-functional requirements. Hopefully, attack patterns were used in the requirement gathering phase to generate security requirements. These security requirements should be tested during system testing. For example, the Unicode Encoding attack pattern can be used to generate test cases that ensure that the application behaves properly when provided with unexpected characters. Testers should provide characters that the application is not supposed to accept to the application to see how it behaves. The application's actual behavior when under attack should be compared with the desired behavior defined in the security requirements.

### *Leveraging Attack Patterns in Regression Testing*

Regression testing is the running of existing tests on the software any time that the code is changed to ensure that the change not only caused the intended behavior but also that it did not inadvertently cause any unintended changes. Attack patterns do not bring any new and unique value to regression testing itself. Effective regression testing should include security test cases developed during the other testing levels that were guided by use of attack patterns.

### *Leveraging Attack Patterns for Testing in the Operational Environment*

Even after application of typical testing levels, software brings with it security concerns applicable to testing. Even if security was considered throughout the SDLC when building software, and even if extensive testing has been performed, vulnerabilities will likely still exist in the software. This is because no useful piece of software is 100 percent secure [Viega 01]. For software to be useful, there must be ways to use it. Revisiting the analogy of a bank vault, a vault could be made extremely secure if it were constructed a few miles underground, was surrounded by several hundred feet of steel-reinforced concrete, had no access doors, and could withstand attacks from nuclear bombs. It might even be 100 percent secure, but it would of course be completely useless. For it to be useful, there must be a way to access it, and an attacker is likely to exploit the access point(s) if the access point(s) are the easiest ways of gaining access. Designers and builders can only ensure that they disallow "side-channel" attacks, so that the only way the attacker can access the vault is through the door built into it. The designers can do absolutely nothing to prevent an authorized employee from giving away the combination to the vault to their friends, from leaving the door ajar, etc. The vault itself can be made extremely secure, but the actual operational environment may be completely insecure. Software also faces similar issues. Software can be designed and developed to be extremely secure, but if it is deployed and operated in an insecure fashion many vulnerabilities can be introduced. For example, a piece of software could provide strong encryption and proper authentication before allowing access to encrypted data, but if an attacker can obtain valid authentication credentials he/she can subvert the software’s security. Nothing is 100 percent secure, and the environment must be secured and monitored to thwart attacks.

Newer object-oriented programming models involving principles such as inversion of control further complicate the problem. For instance, the Spring framework for Java allows components to be “wired” together declaratively, similar to components being assembled together in a car. The entire car does not need to be rebuilt if a manufacturer decides to use a different brand of tires; the Spring framework enables similar swapping of software components during deployment without requiring rebuilding of large pieces. However, the problem is that the system designers and developers may have made assumptions regarding certain components that may not be satisfied by the components that are actually deployed. Such issues cannot be considered the manufacturer’s fault unless they provide insufficient documentation.

Therefore, it is extremely important to perform security testing of the software in its actual operational environment. Vulnerabilities present in software can sometimes be masked by environmental protections such as network firewalls and application firewalls, and environmental conditions can sometimes create new vulnerabilities. Such issues can often be discovered using a mix of white-box and black-box analysis of the deployed environment. White-box analysis of deployed software involves performing security analysis of the software, including its deployed environment, with knowledge of the architecture, design, and implementation of the software. Black-box analysis (typically in the form of penetration testing) involves treating the deployed software as a “black box,” and attempting to attack it without any knowledge of its inner workings. While black-box analysis is relatively inexpensive and can find many of the more obvious and small problems, it is not as effective at finding many of the often more significant issues. These issues are typically found only through in-depth white-box analysis. Black-box is good for finding the specific implementation issues you know to look for, while detailed and structured white-box can uncover unexpected architecture/design and implementation issues that you may not have known to look for. Both types of testing are important, and attack patterns can be leveraged for both.

### *Leveraging Attack Patterns for Black-Box Testing*

Black-box testing of web applications is generally performed using tools such as application security testers like those from companies such as SPI Dynamics that automatically run predefined tests. Attack patterns can be used as models to create the tests these tools perform, thereby giving them more significant relevance and effectiveness. Such tools, though, cannot find many types of architectural flaws, or even all implementation errors. These tools generally test for a large variety of attacks, but they generally cannot find subtle architectural vulnerabilities. They effectively find issues that script kiddies and other relatively unskilled attackers would likely exploit. However, a skilled attacker would be able to find many issues that a vulnerability scanning tool simply could not detect. For instance, a lack of encryption for transmitting social security numbers would not be detected using an automated tool, as the fact that social security numbers are unencrypted is not a purely technical flaw. The black-box testing tool cannot determine what information is a social security number and cannot apply business logic. Attack patterns that are useful for creating black-box tests include those that can be executed remotely without requiring many steps. Some examples of vulnerabilities that black-box testing can detect include cross-site scripting using injection of JavaScript in a HTTP parameter and SQL injection using separator characters. Automated tools can be used to create tests, such as where a separator character is inserted into a HTML form field, to observe whether a database error occurs. Black-box testing of non-web applications can be performed similarly using different tools.

### *Leveraging Attack Patterns for White-Box Testing*

White-box testing is slower but more thorough than black-box. It involves extensive analysis performed by security experts that have access to the software's requirements, architecture, design, and code. The primary goal of white-box security testing is to find the more obscure implementation bugs not found in black-box testing as well as architecture and design flaws and related security issues. The advantage of white-box testing lies in its thoroughness; security experts may analyze a system for several weeks or months while knowing all of its internal details. If the flaws they find are mitigated, it is unlikely that an attacker with limited knowledge of an application's internal workings will easily find a significant vulnerability. Attack patterns can be leveraged to determine areas of system risk and thereby on which areas of the system white-box analysis should focus. The attack patterns most effective for white-box analysis include those that target architecture and design weaknesses. Attack patterns that target specific implementation weaknesses should not be completely disregarded because, in many cases, implementation weaknesses can only be easily found using manual code reviews (a type of white-box analysis). An attack pattern that could be leveraged in white-box testing of a deployed system is sniffing sensitive data on an insecure channel. Those with knowledge of data sensitivity classifications and an understanding of the business context around various types of data can determine if some information that should always be communicated over an encrypted channel is actually sent over an insecure channel. Such issues are often specific to a deployed environment; thus, analysis of the actual deployed software is required.

### *Leveraging Attack Patterns in the Broader Spectrum of Testing*

The testing phase of the SDLC is vital to ensuring the security of the software under development, and, as outlined here, attack patterns can play a valuable and broad role across the various testing activities. There are other more specific types of testing where attack patterns could be leveraged explicitly or implicitly to test the security of the system, but that level of detail exceeds the scope of this paper. Providing such detail is an excellent opportunity for further research and contribution.

## Systems Operation

System operation and attack patterns are related in two ways. First, attack patterns can guide design of secure operational configurations and procedures. Second, operational knowledge of security issues observed in the fielded system can be used to feed back into the attack pattern generation process.

In many cases, software with known vulnerabilities may be deployed because it may be too expensive to fix the problems, no other alternatives may be available, or it may be less expensive to design operational configurations and procedures to react to attacks instead of actually mitigating the issues in the software itself. Having proper operational configurations and procedures in place also is essential, even if software is highly secure. As described in the Leveraging Attack Patterns for Testing in the Operational Environment section above, environmental conditions can dictate whether certain vulnerabilities are present in deployed software, and a large part of environmental conditions consist of operational configurations and procedures. Hence, proper operational configurations and procedures are essential to creating a secure environment [Graff 03].

Attack patterns describe how an attacker may actually exploit software. Given an attack pattern, there may be ways in which certain operational procedures or environmental configurations can thwart the type of attack. For instance, procedures could be put in place to deal with the decompression bomb attack described in the Attack Pattern Generation section until a vendor patch becomes available. Such procedures may include manually deleting or quarantining suspicious e-mails or temporarily blocking all external e-mail access to an organization.

Operations people, with their knowledge of security issues and familiarity with the methods of attackers in an operational environment, can also be a great source for generating new attack patterns. When indications that a system was successfully exploited are present, an investigation that identifies how the attack was carried out is generally conducted. The process described in the Attack Pattern Generation section above can be used during the investigation to potentially generate new attack patterns. These new attack patterns can then be leveraged to modify existing software and/or environmental configurations or to create additional operational procedures for added security.

### Policy and Standard Generation

While, as described in the above sections, attack patterns can certainly be used directly by designers and developers, it is also helpful in many organizations to use attack patterns indirectly during the SDLC by using them to generate policies and standards that are in turn used to develop secure software. These policies and standards can include those generated by third parties, such as with the Payment Card Industry (PCI) standards or those generated for internal use within an organization. Using attack patterns to generate policies is mostly helpful for organizations that need to dictate security standards for other organizations (e.g., credit card consortia, government agencies) and for large software development organizations.

Using policies and standards during the SDLC is not a substitute for the knowledge of attack patterns, and organizations should not rely solely on their software development staff using policies to develop secure software for several reasons. First, it is much easier to concisely describe how software can be abused than to describe how secure software should be built. Mitigations for attack patterns also vary by the technology used. Second, waiting until policies and standards are updated using information from the latest attack patterns adds another layer of indirection that increases the amount of time it takes for software developers to implement countermeasures against the latest attack patterns.

Even though appropriate policies and standards are not substitutes for attack patterns, they are extremely helpful for day-to-day software design and development activities. Policies describe high-level rules that are applicable across all software deployed in an organization. For instance, a policy may state “all data obtained from a network must be sanitized before they are processed by any business logic.” This policy may be designed to address attack patterns such as “command delimiters” and “XSS in HTTP headers.”

Standards are refinements, often seeded with useful examles, of policies that apply to specific software and/or technologies. For example, addressing the attack pattern of “XSS in HTTP headers” in Java Servlets may require use of standards such as “all data obtained from the network that contain characters must have the following characters removed as soon they are seen by a server: ‘<’, ‘>’, ‘(‘, ‘)’, ‘;’ ”. However, it is important to note that standards should always be developed and deployed in a balanced and comprehensive fashion and not in isolation. For instance, the example in the previous sentence applied in isolation could leave a system susceptible to alternate encoding issues and thus should be coordinated and buttressed with other relevant standards as an effective package.

Policies and standards are useful in large organizations because they ensure that mitigations for attack patterns are applied uniformly across all code. Like attack patterns, policies and standards can be used in all phases of the SDLC. Policies and standards also help organizations specify minimal security controls that must be in place for other organizations that handle certain types of data.

### 1.4 Further Information on Attack Patterns

Attack patterns are a rather new concept and, as of yet, relatively little content is available for further reading. The References section below lists some resources that may prove valuable. Specifically, the following resources are directly relevant and should be considered:

The Common Attack Pattern Enumeration and Classification (CAPEC) initiative sponsored by the Department of Homeland Security. The objective of this effort is to develop and deploy to the public an initial baseline catalog of attack patterns along with a comprehensive schema and classification taxonomy. It is hoped that, after its launch (Q2 2007), this catalog will continue to form the standard mechanism for identifying, collecting, refining, and sharing attack patterns among the software community.

- *Exploiting Software: How to Break Code* [Hoglund 04]
- *Attack Modeling for Information Security and Survivability* [Moore 01]
- *Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs* [Gegick 05]

### 1.5 Glossary

**attack** An attack is the act of carrying out an exploit.

**attack path** An attack path is a path in an attack tree from a leaf node to the root node. An attack path can be a simplistic representation of an attack pattern.

**attack pattern** An attack pattern is a general framework for carrying out a particular type of attack such as a particular method for exploiting a buffer overflow or an interposition attack that leverages architectural weaknesses. In this paper, an attack pattern describes the approach used by attackers to generate an exploit against software.

**attack tree** Attack trees (known as “threat trees” by Microsoft) provide a formal, methodical way of describing the security of systems based on various attacks [Schneier 99]. The root node of the tree is the attacker’s goal (known as “threat” by Microsoft), and the “children” of each node describe a lower-level way of achieving the goal of the parent node. In this manner, the leaf nodes generally contain relatively low-level tasks such as “install a key logger on target machine”, and the root node contains a goal such as “obtain administrator’s password.”

**attacker** An attacker is the person that actually executes an attack. Attackers may range from very unskilled individuals leveraging automated attacks developed by others (“script kiddies”) to well-funded government agencies or even large international organized crime syndicates with highly skilled software experts.

**bugs** Bugs are software problems that exist only in code. A bug that exists in code may or may not ever be executed or exploitable. Therefore, a bug may or may not represent a vulnerability in the underlying software. Bugs are used to describe minor implementation errors that are typically easy to fix. Note that simply because bugs are minor implementation errors does not mean that the impact of an attacker exploiting the bug is small. For instance, a buffer overflow is a well-known type of bug that is generally easy to fix. However, exploiting a buffer overflow can give an attacker full control over a system.

**design pattern** A design pattern is a general repeatable solution to a recurring software engineering problem.

**exploit** An exploit is a technique or software code (often in the form of scripts) that takes advantage of a vulnerability or security weakness in a piece of target software.

**flaws** Flaws are software problems that exist in the software’s design. A flaw may or may not represent a vulnerability in the underlying software. Mitigating a flaw typically involves significantly more effort than simply modifying a few lines of code. The problem does not lie solely in the implementation; the underlying design is flawed, and therefore, any implementation that follows the design would contain the flaw. For instance, performing sensitive business logic in an untrusted client application is a design flaw that cannot be mitigated by a simple measure such as modifying array bounds.

**impact** Impact is the effect that the organization using vulnerable software faces if a vulnerability were to be exploited. Impact could range from specific tangible values such as monetary fines from the breach of a law or regulation to intangible values such as brand and reputation damage.

**misuse/abuse cases** Misuse/Abuse cases can be viewed as use case requirements with an attacker as the actor. They represent actions taken against the software that do not fall within the normal, defined operating parameters of the system. They are typically identified and modeled in an integrated fashion with positive functional use cases for a system.

**risk** Risks capture the likelihood that a vulnerability will be exploited, as well as the potential damage (impact) that will occur if it is. It is important to note that risks, threats, and exploits are all separate things. Risks may be present in the target software, on the target host, or in the broader operational environment of the software.

**risk analysis** Risk analysis involves analyzing target software for vulnerabilities and characterizing their nature and potential impact. Microsoft calls this “threat modeling.” Risk analysis attempts to identify, prioritize, and plan appropriate mitigation for the risks facing a piece of software.

**security pattern** A security pattern is a design pattern that is intended to show software developers how to design and implement solutions to common security problems. These solutions typically represent software security features. A security pattern may be used to mitigate multiple attack patterns, and an attack pattern may be mitigated using multiple security patterns.

**target host** A target host is the computer or platform that is running the target software of an attack. A host may be attacked through interfaces of the target software or through purely network-based attack mechanisms.

**target software** Target software is software that is the target of an attack.

**threat** A threat is an actor or an agent that is a source of danger to the system under consideration or the assets to which it has access. The threat can be a person that abuses the software, a program running on a compromised system, or even a non-sentient event such as a hardware failure. A threat exploits a vulnerability in software to attack it.

**vulnerabilities** A vulnerability is a software weakness that can be exploited by an attacker. Bugs and flaws collectively form the basis of most software vulnerabilities.

**weakness** A weakness is an underlying condition or construct existing in a software system that has the potential for negatively impacting the security of the system.

## 1.6 References

[Alexander 64]

Alexander, Christopher. *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press, 1964.

[Alexander 77]

Alexander, Christopher; Ishikawa, Sara; & Silverstein, Murray. *A Pattern Language*. New York, NY: Oxford University Press, 1977.

[Alexander 79]

Alexander, Christopher. *A Timeless Way of Building*. New York, NY: Oxford University Press, 1979.

[DOA 88]

Department of the Army. *AR 380-5 Department of the Army Information Security Program, Classified Document and Materiel Storage* (1988).

[Ellison 06]

Ellison, Robert. *Attack Trees*. (2006).

[Gamma 95]

Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995.

[Gegick 05]

Gegick, Michael & Williams, Laurie. “Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs.” *ACM SIGSOFT Software Engineering Notes, Proceedings of the 2005 workshop on Software engineering for secure systems—building trustworthy applications SESS '05, Volume 30, Issue 4*. New York, NY: ACM Press, 2005.

[Graff 03]:Graff, Mark & van Wyk, Kenneth. *Secure Coding: Principles and Practices*. Sebastopol, CA: O’Reilly and Associates, 2003.

[Hoglund 04]

Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley, 2004 (ISBN 0-2017-8695-8).

[Howard 02]

Howard, M.; & LeBlanc, D. *Writing Secure Code*. Redmond, WA: Microsoft Press, 2002.

[Kienzle 01]

Kienzle, Darrell & Elder, Matthew. *Security Patterns* (2001).

[Koizol 04]

Koizol, Jack; Litchfield, D.; Aitel, D.; Anley, C.; Eren, S.; Mehta, N.; & Riley. H. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Indianapolis, IN: Wiley, 2004 (ISBN 0764544683).

[Leveson 83]

Leveson, Nancy G. & Stolzy, Janice L. “Safety analysis of ada programs using fault trees.” *IEEE Transactions on Reliability R-32*, 5 (December 1983): 479-484.

[Leveson 04]

Leveson, Nancy. “A Systems-Theoretic Approach to Safety in Software-Intensive Systems.” *IEEE Transactions on Dependable and Secure Computing 1*, 1 (January-March 2004): 66-86.

[McGraw 06]

McGraw, Gary. *Software Security: Building Security In*. Boston, MA: Addison-Wesley, 2006. http://www.buildingsecurityin.com

[Moore 01]

Moore, A. P.; Ellison, R. J.; & Linger, R. C. *Attack Modeling for Information Security and Survivability* (CMU/SEI-2001-TN-001, ADA388771). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001.

[ReliaSoft 03]

ReliaSoft. *Fault Tree Analysis, Reliability Block Diagrams and the BlockSim FTI Edition*, 2003.

[Schneier 99]

Schneier, Bruce. “Attack Trees: Modeling Security Threats.” *Dr. Dobb’s Journal*, December, 1999.

[Schumacher 06a]

Schumacher, M.; Fernandez-Buglioni, E.; Hybertson, D.; Buschmann, F. & Sommerlad, P. *Security Patterns: Integrating Security and Systems Engineering*. New York, NY: John Wiley & Sons, 2006.

[Schumacher 06b]

Schumacher, Markus. *SecurityPatterns.Org*. (2006).

[Swiderski 04]

Swiderski, F. & Snyder, W. *Threat Modeling*. Redmond, WA: Microsoft Press (2004).

[Vesely 81]

Vesely, W. E.; Goldberg, F. F.; Roberts, N. H.; & Haasl, D. H. *Fault Tree Handbook (NUREG-0492)*. Washington, DC: Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission, 1981.

[Viega 01]

Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2001.

[Whittaker 03]

Whittaker, James. *How to Break Software Security: Effective Techniques for Security Testing*. Boston, MA: Addison-Wesley, 2003.