Towards a Specification Prototype for Hierarchy-Driven Attack Patterns

Joshua J. Pauli  
College of Business and Information Systems  
Dakota State University  
Madison, SD, 57042, USA  
josh.pauli@dsu.edu

Patrick H. Engebretson  
College of Business and Information Systems  
Dakota State University  
Madison, SD, 57042, USA  
pat.engebretson@dsu.edu

Abstract

We propose the characteristics of a software tool that leverages specifying attack pattern details in understandable hierarchies. These hierarchies are currently manually populated from the vast CAPEC dictionary which consume an excessive amount of human resources and are wrought with the possibility of user error. Such a software tool will not only automate the population of these attack pattern hierarchies, but also provide system prerequisite information and suggested mitigation strategies for the system under design. The combination of system prerequisites, possible attack patterns, and necessary mitigation strategies gives system designers and developers a checklist-like artifact to consider as development moves from the design phase to the implementation phase. This artifact is valuable because the “patch and pray” mentality of software security is insufficient. This prototype tool is also intended to generate managerial-level policy documentation that can be used in the justification and enforcement of development standards in any given environment. Because our prototype is part of the design phase of development, it must be configurable for many system-specific choices such as, but not limited to, operating system, hardware, programming language, and data access and storage techniques. The importance and significance of our approach is justified by leveraging the recently released CAPEC dictionary as part of development for systems that have a strong sense of security mandated.

Keywords: Attack Trees, Attack Patterns, Refinement, Hierarchy.

1. Introduction

Within the past five years, the fields of network, computer, and software security has begun to shift its focus away from perimeter defensive models, such as border routers, firewalls, and intrusion detection systems, to more proactive defensive models [10]. Until recently many companies have simply relied on a patch-when-exploited methodology to writing secure software [7]. This patch and penetrate methodology does nothing to address the underlying security issues. In order to better instantiate a proactive defense model, one must include software security and make sure that these priorities are carried throughout every phase of the software development lifecycle. Good security and the ability to combat malicious code is the byproduct of understanding said code’s mechanics as well as its motivations [3].

Our prototype tool takes the proactive mindset and attempts to automate the populating of attack trees for the benefit of security-centric design decisions. Because security strategies vary greatly after design decisions are made, our prototype includes built-in mappings for many well established design configurations as well as the ability to add custom design configurations. With these configurations established, the tool can aid in the development of secure systems by creating a checklist-like artifact that the development team can follow to ensure that necessary mitigation strategies are implemented to account for the known attacks against the selected design configuration.

The mappings within our tool are derived from the CAPEC dictionary of attack pattern information. Attack Patterns are relatively new, having been introduced within the past decade [13]. It is the goal of this paper to leverage this vast repository of attack pattern information with an automated tool that creates usable hierarchies and mappings for security-based systems.

Section 2 covers related work that our prototype is based on. Section 3 introduces our attack pattern hierarchy model. Section 4 introduces the characteristics of our prototype tool. Section 5 covers future work and we conclude in section 6.
2. Related Work

Fostering a deep and solicitous understanding of attack patterns can lead to the permeation of security throughout the software development life cycle, as well as heighten awareness of known exploits, vulnerabilities and weaknesses [9]. Integrating security throughout the software development lifecycle has long been understood as being very important [10], but practical teaching methods have been difficult to specify and implement. Integrating attack pattern knowledge into the SDLC can result in more secure software by creating less exposure to identified bugs and known flaws [10]. Attack patterns can be used to create a security checklist for developers, which in turn can lead to more secure software [4].

The origins of attack patterns can be traced back to concepts outlined by Gamma, et. al. when the foundation for today’s attack patterns where established as the concept of a general, repeatable solution to identified system development problems [8]. More recently the concept of presenting from an attacker’s perspective was done on an individual, or attack by attack basis, with no agreed upon formula, structure, or common language for consistently presenting such a viewpoint [10].

The lack of a common or united vocabulary makes it difficult to gather, analyze, and share pertinent information in meaningful ways which could be used to advance the discipline of software security. Moore began to formalize a concept of combining various types of malicious attacks (i.e. the attacker’s perspective) with the pattern framework [8]. Hoglund and McGraw built upon the foundation of Moore’s paper by more formally defining attack patterns and identifying 48 distinct attack patterns [10].

The final piece of the attack pattern puzzle was realized when the National Cyber Security Division of the Department of Homeland Security in conjunction with Cigital and MITRE Corporation agreed to sponsor CAPEC. The final result of this collective effort was published in March of 2007 and included a formalized attack driven perspective of software security with 101 different attack patterns outlined [6].

The Common Attack Pattern Enumeration and Classification (CAPEC) list provides an official schema and formal representation for defining attack patterns [6]. Release 1 of CAPEC included roughly 100 attack patterns, which tends to be too many attack patterns to effectively use in the classroom setting. CAPEC formally organizes and presents each attack pattern by gathering and displaying both primary and supporting data elements [5].

Primary elements include:

- Attack Pattern ID
- Attack Pattern Name
- Description
- Related Weaknesses
- Related Vulnerabilities
- Methods of Attack
- Examples-Instances
- References
- Solutions and Mitigations
- Typical Severity
- Typical Likelihood of Exploit
- Attack Prerequisites
- Attacker Skill or
- Knowledge Requirements
- Resources Required
- Attack Motivation-Consequences
- Context Description

Supporting elements include:

- Injection Vector
- Payload
- Activation Zone
- Payload Activation Impact
- Probing Techniques
- Indicators/Warnings of Attack
- Obfuscation Techniques
- Related Attack Patterns
- Relevant Security Requirements
- Relevant Design Patterns
- Relevant Security Principles
- Related Guidelines

Numerous approaches in the requirements phase have addressed software security including scenario-driven requirements analysis [2], misuse case-driven elicitation of non-functional requirements [1][15] goal-driven approaches to user requirements [11], abuse-frame oriented security requirements analysis [12], anti-goals [17], abuse frames, and reuse-based approaches to determining security requirements [16]. The anti-goal method in the goal-oriented requirements analysis can describe vulnerabilities or generate attacks that violate security goals [17]. The “actor, intention, goal” approach in the i* framework [18] models security and trust relationships as “software goals”: goals depending on another actor to satisfy them.
3. Hierarchy-Driven Model

The CAPEC list provides a common set of elements and possible values to ensure consistency among attack patterns. Because of the vast quantities of CAPEC (100 attack patterns and supporting elements), it is unrealistic to be able to manually make sense of all the information.

Given the significance of the subject matter and importance of such concepts, we initially developed our approach for teaching attack patterns. Our model relies on utilizing a hierarchy to present attack pattern information logically. We focus on a slim element set to incorporate into our hierarchy model [10].

We utilize a top-down approach with the highest level being the most general element. Subsequent hierarchy levels will become more specific in nature and scope following a minimum 1:1 ratio for each level of the hierarchy as refinement continues from one level to the next.

Our hierarchy model is introduced in Figure 1 where more details are realized about the attack pattern as refinement continues to subsequent levels of abstraction. Vulnerabilities (top) effectively group the attack patterns in understandable contexts. The CAPEC dictionary element that each level is derived from is listed to ensure consistent mapping from the CAPEC dictionary to the attack pattern hierarchy.

Level 1: Vulnerability: A general classification used to group a collection of related errors that an attacker can exploit. Every attack pattern outlined by CAPEC is a child of one of the following vulnerabilities: Abuse of Functionality, Spoofing, Probabilistic Techniques, Exploitation of Authentication, Resource Depletion, Exploitation of Privilege/Trust, Injection, Data Structure Attacks, Data Leakage Attacks, Resource Manipulation, Protocol Manipulation, and Time State Attacks.

Level 2: Attack Pattern: A high level blueprint that describes various types of software attacks.

Level 3: Exploit: A specific instance of an attack pattern. Level 3.1 Bug / Flaw is used to explain difference between a design issue (flaw) and an implementation issue (bug). Level 3.1 has been added to force deep thinking about the type of exploit.

Level 4: Activation Zone: The area in a software package which is capable of activating or executing a payload or exploit.

Level 5: Injection Vector: The actual format of the input used in an attack. Level 5.1: Payload references any input given to the software in order to carry out an exploit. The hierarchy allows for the fact that not every Attack Pattern makes use of a payload.

Level 6: Reward: The output event or desired outcome of a successful exploit.

Figure 1 introduces one “Server Side Include (SSI) Injection” attack pattern in our hierarchy.

Because of the hierarchy format it is now known what elements influence other elements in the same attack pattern. We use existing and accepted definitions for each abstraction level in our hierarchy [10].

Figure 1. Levels Mapped to CAPEC Dictionary

Figure 2. Populated Hierarchy for “SSI Injection” Attack Pattern
4. Tool Prototype Characteristics

Our prototype tool is based on the CAPEC dictionary to ensure that the input, processing, and output are derived from an accepted source [6]. Each attack pattern contains all the relevant information to populate our prototype tool’s data store for later retrieval. This information is then organized and extracted to be useful for the design and implementation teams as the development process moves forward. The goal of the prototype tool is to provide a checklist-like artifact to ensure security is considered as part of design and long before the implementation phase.

The input for our prototype tool is the prerequisites from the CAPEC dictionary; these are design decisions that are made for the system under design. Our prototype accepts prerequisites such as hardware selections, operating system, server configurations, and programming language used for initial input. As a user selects a system prerequisite, related attack patterns and necessary mitigation strategies are populated to be reviewed.

The processing of our prototype tool is comprised of extracting, organizing, and editing data mappings that are made up of system prerequisites, related attack patterns, and necessary mitigation strategies. The data mappings are stored in a database and are leveraged by extracting and presenting them at a system-specific level. The user of the prototype tool can edit any mapping between prerequisite and attack pattern or between attack pattern and mitigation strategy to best satisfy their system’s requirements and configurations.

The output of our prototype tool is twofold. First, there is organized output as part of the normal usage of the tool that displays the system-specific prerequisites, related attack patterns, and necessary mitigation strategies in a hierarchical format. Any part of the output can be edited to better reflect the exact nature of the specific system. This output is available graphically, which is most applicable for small systems. Second, these mappings can also be viewed in tabular format and managerial reports, which is most applicable for larger systems.

4.1. Hierarchy-Driven Data Mappings

Our prototype tool is driven by stored data mappings of system prerequisite, related attack patterns, and necessary mitigation strategies for a specific system. We developed the master set of data mappings as best practices for many common system prerequisites per the CAPEC dictionary [6]. Because of the vast amount of information in this dictionary, our master set of data mappings will be incrementally populated as more systems are developed using our tool.

Our prototype tool also allows for data mappings to be added, edited, and/or deleted from the system’s specifications. Because every system has its own set of data mappings, the master set of attack pattern-driven data mappings are left unchanged. This allows for the design and implementation teams to truly customize the system’s security components to best match the system.

A system’s information is stored in a separate set of tables that are derived directly from the master data mappings. We call this a “system profile”. The prototype tool extracts and lists the master data mappings per the selected system prerequisites. They are listed in a tree structure as introduced in Figure 3.

![Figure 3. Abstract Tree Structure for Displaying System Prerequisite-Driven Data Mappings](image)

Once these default data mappings are in the tool, they can be added to or edited to best suit the system under design. Each node is expandable because of the tree structure to show all of the “downstream” entities that are related. That is, the system prerequisite is expandable to show all of the attack patterns that threaten it, while the attack pattern is expandable to show all the necessary mitigation strategies that may be used to prevent it or minimize its damage. One issue with systems of any size is that the tree structure quickly becomes unreadable because of the number of links. Because of this issue, the mappings for a specific system may be best viewed in a tabular format. Table 1 introduces a partially populated hierarchy for the “Apache Webserver” system prerequisite.
### Table 1. Partial Hierarchy for the “Apache Webserver” System Prerequisite

<table>
<thead>
<tr>
<th>PREREQ</th>
<th>ATTACK</th>
<th>MITIGATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Webserver</td>
<td>Server Side Include (SSI) Injection</td>
<td>Set the OPTIONS IncludesNOEXEC in the global access.conf file or local .htaccess (Apache) file to deny SSI execution in directories that do not need them</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All user controllable input must be appropriately sanitized before use in the application</td>
</tr>
<tr>
<td></td>
<td>HTTP Request Smuggling</td>
<td>Careful analysis of the entities must occur during system design prior to deployment. If there are known differences in the way the entities parse HTTP requests, the choice of entities needs consideration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Employ an application firewall</td>
</tr>
<tr>
<td></td>
<td>Session Credential Falsification through Prediction</td>
<td>Use a strong source of randomness to generate a session ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use adequate length session IDs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do not use information available to the user in order to generate session ID (e.g., time)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ideas for creating random numbers are offered by Eastlake [RFC1750]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encrypt the session ID if you expose it to the user.</td>
</tr>
</tbody>
</table>

As shown in Table 1, applicable attack patterns for “Apache Webserver” are “Server Side Include Injection”, “HTTP Request Smuggling”, and “Session Credential Falsification through Prediction”. Specific mitigation strategies for each of these attack patterns are presented in column three of Table 1. These mappings are originally stored as part of the CAPEC-driven master data mappings, and can be edited at the discretion of the development team to best reflect the exact implementation of the system.

Greater details about the system prerequisite, related attack patterns, and the necessary mitigation strategies can be accessed as part of our prototype. All of this information is stored as part of the system-specific data mappings. Such information will include, but is not limited to, the following database table entities.

- **System Prerequisite:**
  - System ID
  - System Name
  - System Category (e.g. operating system, hardware, programming language, data access, storage techniques).

- **Attack Pattern:**
  - CAPEC Attack Pattern ID
  - CAPEC Attack Pattern Name

- **Mitigation Strategy:**
  - Mitigation ID
  - Mitigation Name
  - Mitigation Description

### 4.2. Tool Prototype Output

The graphical representation of the system-specific data mappings is not ideal for use later in the development process. This graphical representation is effective primarily in displaying the links between the system prerequisites, related attack patterns, and necessary mitigation strategies. This also provides an effective medium to edit any of these links per the system design and implementation details.

The real usability of our prototype tool comes from the ability to export the tabular format of the system-specific data mappings as a report. The table (similar to Table 1) itself can be exported, which is much more readable than the graphical representation. Furthermore, a more formalized report can be generated from the prototype tool that can be used as policy for development managers to mandate how systems are to be developed. This is especially useful for systems that frequently share similar hardware, operating system configurations, webserver configurations, and other related system prerequisite decisions.
5. Future Work

The immediate future work for this approach is the actual creation of a tool that embodies the characteristics introduced and satisfies a set of specified functional and non-functional requirements. One area of special importance is the ratio of elements as refinement continues and more details are known about each attack pattern. Such a software tool would eliminate human errors and greatly increase the processing time in populating the hierarchy.

We recognize that not every attack pattern defined in CAPEC (Release 1) fits neatly into the proposed hierarchy. The most common missing CAPEC elements include the following: Activation Zone, Injection Vector, and Payload. We must address this in the creation of the software tool. The definitions and differences between a bug and a flaw must also be adequately defined and dealt with by the software tool.

6. Conclusions

Developers who are presented with vast amounts of information need assistance to avoid feeling overwhelmed and frustrated [14]. This is the reason for the creation of a characteristic set for a software tool that makes use of the CAPEC dictionary of attack pattern information. Because of the exhaustive nature of the CAPEC dictionary, developers and researchers can easily feel overwhelmed by the sheer amount of usable information. Our tool is an attempt to automate this information into a set of accepted and usable data mappings that are used to populate attack pattern hierarchies. The inclusion of system prerequisites, likely attack patterns, and necessary mitigation strategies ensure that the vast CAPEC dictionary is leveraged during the design phase of secure-centric software. The hierarchical nature of our approach allows for developers to easily comprehend the different levels of abstraction and how they are related to one another. The artifacts of our tool can then be used to ensure development follows established development standards.

References