# From Vulnerability Anti-Patterns to Secure Design Patterns

Alok Chandrakant Ratnaparkhi
*Dept. of Computer & Information Science*
*University of Massachusetts Dartmouth*
Dartmouth, MA, U.S.A.
aratnaparkhi@umassd.edu

Onyeka Ezenwoye
*Computer and Cyber Sciences*
*Augusta University*
Augusta, GA, USA
oezenwoye@augusta.edu

Yi Liu
*Dept. of Computer & Information Science*
*University of Massachusetts Dartmouth*
Dartmouth, MA, U.S.A.
yliu11@umassd.edu

***Abstract*—A secure design pattern is a well-proven reusable solution to a recurring security problem that arise in specific contexts. Using secure design patterns properly can help tackle software vulnerabilities during software development. However, the lack of selection guidance of secure patterns makes it more difficult for developers to use secure design patterns than conventional design patterns. To address this issue, this paper presents a methodology of selecting the appropriate secure design patterns for software vulnerabilities formalized in anti-patterns. This methodology bridges the gap between the vulnerabilities and secure design patterns to produce a useful tool for secure software development.**

***Index Terms*—software vulnerabilities, anti-patterns, secure design patterns**

## I. Introduction

Software vulnerabilities are weaknesses in a system's architecture, design, or code that can cause violations of the system's security policy. Its existence is the primary cause of attacks on software systems [1]. Even though security is a critical quality attribute, security is often seen as an afterthought. Many research studies on software vulnerabilities focus on testing methodologies, such as using machine learning [2] to identify the common vulnerabilities.

Although software testing is absolutely mandatory and necessary in locating the vulnerabilities, security needs should be emphasized throughout the entire software development process and the vulnerabilities should be identified as early as possible. After software has been deployed, it becomes more expensive to remove vulnerabilities by patching. Thus, software developers should elicit security expectations during the requirements analysis stage and consider them during the stages that follow. Citing research by Software Engineering Institute (SEI), the U.S. Department of Homeland Security (DHS) states in its Software Assurance information sheet that "90% of reported security incidents result from exploits against defects in the design or code of software" [4]. Most of the vulnerabilities reported in software are the results of bad decisions made during the implementation stage, however, many of them actually originate in the previous software development stage, the *design* stage. "Secure by design" is not a new concept and has been explored by many researchers. Deogun et al. [5] introduced the good practices for implementing essential software features using design as the primary driver for security. Several researchers have published various secure design patterns to address security vulnerabilities at the design level [8], [9]. However, very few research studies have explored the methodology of selecting appropriate secure design patterns to mitigate security vulnerabilities.

This research proposes a methodology for selecting the appropriate secure design patterns for addressing software security vulnerabilities. The rest of the paper is organized as follows. Section 2 briefly discusses the security vulnerabilities, secure design patterns, and anti-patterns. Section 3 presents the methodology of selecting appropriate secure design patterns for addressing the security vulnerabilities. Section 4 discusses the related works, and section 5 concludes the work.

## II. Background

### A. *Software Vulnerabilities*

A software vulnerability is an exploitable flaw in any part of a system's artifact or component. Cross-site scripting (XSS) is an example of a commonly recurring security vulnerability in web applications. With the advent of scripting languages such as JavaScript, it has become feasible for attackers to inject malicious scripts into a victim's browser. By injecting the script, attackers can read private data, hijack the user session, or delete essential data. Fig.1, 2 & 3 show an example of a *DOM-based XSS attack*, which is found in OWASP's Juice Shop application [13]. The application's *search product* function is intentionally developed in such a way that user input is not validated or sanitized before execution. An attacker can enter malicious script (Fig.1) as user input in the search field. The malicious script entered into the search field (Fig. 2) is then submitted to the server and gets executed on the client side. In this example, code from a different domain (soundcloud.com), in the form of a soundtrack, is inserted instead of legitimate search results (Fig. 3).

### B. *Anti-Patterns*

Anti-patterns are commonly occurring solutions to a problem that generates negative consequences [3]. Software development is a complicated process and many decisions may cause a project to fail. Formal documentation of the decisions

DOI reference number: 10.18293/SEKE2021-179

```
<iframe width=" 100% " height="166" scrolling ="no"
frameborder ="no" allow =" autoplay " src =" https://w.soundcloud.com/
player /? url= https %3A// api.soundcloud.com/ tracks /771984076& color
=%23 ff5500 & auto_play = true & hide_related = false & show_comments =
true & show_user = true & show_reposts = false & show_teaser = true " >
</ iframe >
```

Fig. 1. XSS malicious script

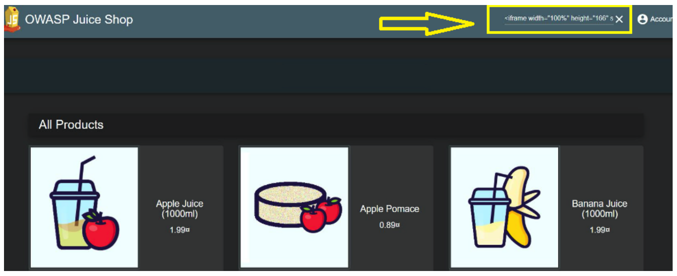

Fig. 2. The malicious script is entered into the search field

or processes that lead to failure can guide future software engineers on what to avoid. Anti-patterns can help developers capture the causes of security vulnerabilities in architecture, design, or source code. Dougherty [8] discussed the need for anti-patterns to describe bad decisions, causing software security failures in a formal way. Nafees et al. [12] proposed the format of documenting vulnerability into anti-patterns.

### C. Secure Design Patterns

Secure design patterns help to prevent the occurrence of vulnerabilities [8]. Secure design patterns address security issues in the architectural design and implementation phases of the development life cycle [8]. Many secure patterns have been proposed and a good number of significant work have been published [8], [9].

## III. Methodology

The approach we designed for selecting secure design patterns to address security vulnerabilities include 3 steps: 1) formalizing a software vulnerability using anti-pattern description model; 2) selecting the secure design pattern candidates that can address the vulnerability's anti-pattern; and 3) testing the selected secure design pattern(s) for the anti-pattern problem.

### A. Formalizing Vulnerability to Anti-Pattern Description

The first step is to identify and formalize the vulnerability using an anti-pattern description. We adopt the anti-pattern description model [12] with the following modifications:

- Our model follows conventional anti-pattern description model to include elements *Context*, *Problem*, *Solution*, and *Consequences*. These elements are important to help the audience understand what the actual vulnerability is.
- Our model introduces the element *Root Causes*, which is an essential factor in identifying appropriate solutions.

The elements *Problem*, *Context*, *Root causes*, and *Solution* are the major factors to address in mapping the anti-pattern to appropriate secure design patterns. Our vulnerability anti-pattern description model has ten elements. The following is an

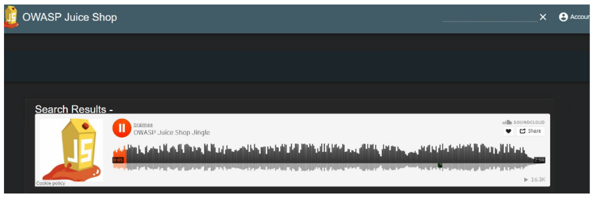

Fig. 3. The malicious script injects code from another domain when submitted

example of using the anti-pattern description model to specify Cross-Site Scripting (XSS) anti-pattern.

1) **Anti-pattern Name:** Cross-Site Scripting Anti-pattern
2) **Also Known as:** Improper Neutralization of Input During Web Page Generation, XSS
3) **Context:** Any web application that uses JavaScript, VBScript, ActiveX, CSS, or any other scripting language
4) **Anti-pattern problem:** Cross-site scripting (XSS) is an injection attack in which attackers can execute malicious scripts on the victim's browser.
5) **Root causes:**
   • *Lack of input validation:* Vulnerable applications trust the data from input without validating it. Without validation, malicious content can be executed in the victim's browser.
   • *Lack of data sanitization:* The victim's browser misinterprets external malicious data as a part of the script and executes it without sanitization.
6) **Example:** A *DOM-based XSS attack* example is found in OWASP's Juice Shop application [13], as shown in Section II-A.
7) **Consequences:** Private data breach; User session hijacking; Identity theft; Phishing Attacks; Web site defacement; Port scan; Keylogging; Trojan Attack
8) **Solution:**
   •*Data sanitization:* Sanitize the external inputs via encoding or escaping. The encoding must be applied to all potential vectors.
   •*Input validation:* Validate the inputs using *blacklist* or *whitelist* validation (preferable).
   •*Miscellaneous solutions:* Using process level, technology-specific or configurational solutions, such as implementing Content Security Policy, using HTTPOnly cookie flag, SameSite cookie parameter, etc.
9) **Attack Types:** Reflected/non-persistent XSS attack, Stored/persistent XSS attack DOM-based XSS attack, Self-XSS, Mutated XSS, and Universal XSS attack
10) **Common Weakness Enumeration:** CWE-79 [18]

### B. Selecting secure patterns for addressing the anti-pattern

The second step of the approach is to go through the pool of secure design patterns and select the best-fit candidates that can address a vulnerability anti-pattern. Two phases, *Collection Phase* and *Analysis Phase*, are involved in this step.

*1) Collection Phase:* Secure patterns are often published in conferences, academic literature, books,repositories, and the internet [8], [9]. The collection phase begins with searching for published secure design patterns. The most crucial problem in the pattern community is, there is no single comprehensive secure pattern repository that exists today. This step can be skipped after a secure pattern repository is developed.

Below are the samples from our collected secure patterns for addressing the XSS anti-pattern. They are categorized into architecture, design and implementation levels.

• Architectural level: *Application Firewall, Broker, Roles* [15]
• Design level: *Secure Strategy Factory, Secure Chain of Responsibility, Intercepting Filter, Controlled Object Monitor,Secure Logger* [8]
• Implementation level: *Account lockout, Client Input Filters, Input Validation* [8]

*2) Analysis Phase:* The *analysis phase* is to analyze the collected patterns from the previous phase. The approach is to find potential solutions by mapping the elements *Root causes, Problem, Context, Solution* of a vulnerability anti-pattern description to the essential elements of secure design patterns. These elements include *Intent, Problem, Context, Forces, Motivation, Applicability, Solution,Structure/Participants, and Collaborations*. The following questions should be asked and answered during the mapping process:

1) Question1: Does the *Intent* element is present or sufficient in capturing the pattern's purpose?
2) Question2: Does the *Intent* of the secure pattern help address the vulnerability anti-pattern's root cause(s)?
3) Question3: In case the *Intent* element is not present or insufficient in capturing the purpose of the pattern, Do *Problem/Context/Forces/Motivation/Applicability* are relevant to the vulnerability anti-pattern's root cause(s)?
4) Question4: Does *Solution/Structure/Participants and Collaborations* of the secure pattern help deliver the *Solution* of the anti-pattern?

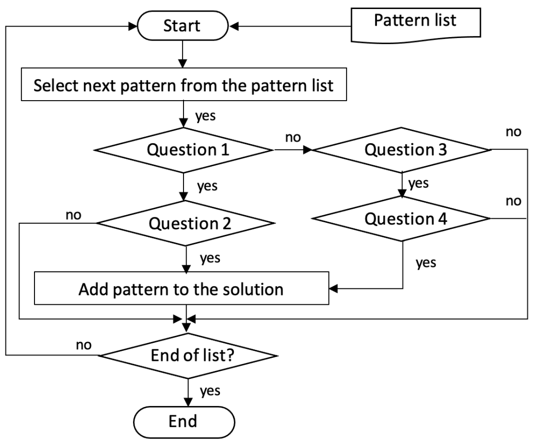

Fig. 4. Flow chart of Mapping Process

The workflow of the mapping process is illustrated in Fig. 4. The analysis begins with scanning of the pattern description model's *Intent* element to understand the purpose of the pattern and check if it can map to the root cause(s) of the targeted vulnerability anti-pattern. If the *Intent* element is absent or insufficient, explore other elements (such as *forces*) of the secure pattern description model. If the secure pattern's *Intent* or solution does not address the root cause, the pattern is ruled out; otherwise, add the pattern in the selected pattern list.

TABLE I
SECURE DESIGN PATTERNS AND THEIR INTENT

| **Secure Patterns** | **Intent** |
|---|---|
| Application Firewall [A] | To filter calls and responses to/from enterprise applications, based on an institution access control policy |
| Broker pattern [A] | Coordinates communication between client and server via requests and responses |
| Roles [A] | Organizing the users with similar security privileges |
| Controlled Object Monitor[D] | To control access to objects by processes |
| Intercepting Filter [D] | To provide mechanism for centrally intercepting the requests and pass them through series of filters before forwarding them to intended destination |
| Secure Chain of Responsibility[D] | To preprocess or postprocess requests/responses using series of handlers |
| Secure Logger [D] | To facilitate in centralized logging mechanisms |
| Secure Strategy Factory [D] | To facilitate easy creation of security objects and use of interchangeable security strategies |
| Account Lockout [I] | Lock the user's account after limited number of incorrect password attempts |
| Client Input Filters [I] | All incoming requests from the client should be filtered at the server. |
| Input Validation [I] | To validate all external inputs from untrusted data sources |

The mapping process of secure patterns' *Intent* to XSS anti-pattern's root causes are illustrated in Table 1. It lists the *Intent* of each of the collected secure design patterns from the *Collection phase*. The intent of each secure pattern is then compared with the root cause of the XSS anti-pattern to find a mapping. Table II gives the reasoning of whether a secure pattern is considered fitting to solve the anti-pattern's root causes. Several secure patterns are ruled out at this point.

Same strategy is used in the analysis of other collected secure patterns. Due to the page limit, the complete analysis process is not presented. Table III presents the selected secure patterns to address the XSS anti-pattern without considering additional requirements on the solution space, such as performance and flexibility. However, additional requirements or attributes required in the design are not avoidable. They should be carefully analyzed in this phase.

### *C. Testing the suggested secure design patterns*

The final step is to apply the selected secure design pattern on the already vulnerable application to test whether the secure design patterns can tackle the anti-pattern problem. Any false positives from the final list are eliminated at this stage. We redesigned the OWASP Juice Shop [13] with *Secure Strategy Factory* pattern, *Input Validation* pattern and *Intercepting Filter* pattern at the design and implementation level [17]. Manual security testing was performed on [13]. The malicious script shown in Fig. 1 was not executed by browser after the redesign. Results show that the selected secure design patterns can be adopted to address XSS anti-pattern problem.

TABLE II
SECURE PATTERN INTENT VS. ROOT CAUSE(S) OF XSS ANTI-PATTERN

| Secure Pattern | Addressing root causes of XSS Anti-pattern ? |
|---|---|
| Application Firewall[A] | The pattern supports the usage of firewalls to detect possible attacks by scanning for malicious signatures using Input Validation. It is most relevant to our anti-pattern problem. |
| Broker pattern[A] | This pattern may help if a separate component is placed between client and server to validate or sanitize the external data. |
| Roles [A] | **Ruled out:** The pattern is related to organizing the users with same role, which is irrelevant to the anti-pattern problem. |
| Controlled Object Monitor[D] | **Ruled out:** This pattern is for controlling objects, which is irrelevant to our anti-pattern problem |
| Intercepting Filter[D] | The pattern may be used to intercept all web requests and pass them through filters to eliminate malicious requests using Input Validation. |
| Secure Chain of Responsibility[D] | May be used for preprocessing the web requests using Input Validation and/or Sanitization. |
| Secure Logger [D] | **Ruled out:** Logging can not help in addressing the anti-pattern problem |
| Secure Strategy Factory [D] | The pattern allows interchangeable strategies and separate them from client who uses it. May help in designing Input Validation/ Sanitization strategies |
| Account Lockout [I] | **Ruled out:** The pattern is related to authentication, which does not help in Input Validation/Sanitization. |
| Client Input Filters [I] | This pattern may help if the good inputs are filtered from bad inputs using Input Validation |
| Input Validation[I] | This pattern is most relevant to our anti-pattern problem. |

TABLE III
MAPPING ANTI-PATTERNS TO SECURE DESIGN PATTERNS

| Vulnerability Anti-Pattern | Secure Design Pattern | | |
|---|---|---|---|
| | *Architectural* | *Design* | *Implementation* |
| XSS Anti-pattern | Broker; Application Firewall | Intercepting Filter; Secure Strategy Factory; Secure Chain Of Responsibility; | Input Validation; Client Input Filters |

## IV. DISCUSSION

The research on security pattern is an active and growing field across the globe [14]. Much research has been done on the secure design pattern classification, but little research has considered selecting appropriate secure design patterns for a given problem. The pattern selection approach proposed in [15] was generalized and not intended for security goals. Alvi et al. [1] proposed a security pattern selection technique based on security objectives and security flaws. However, security flaws do not formally capture what bad decisions can cause the vulnerability in the applications.Different from Alvi's [1] approach, our methodology is based on the anti-pattern model, which presents not only the result but the causes of a vulnerability. Our research does not aim for suggesting the best pattern for a given context like the other approaches [1], [16], but on selecting potential pattern(s) for an anti-pattern problem. In addition to the secure patterns discussed in this paper, other patterns, such as the process patterns and technology patterns [3], can also be used to address vulnerability anti-pattern problems. Our methodology can be easily extended to involve such patterns in the solution.

## V. CONCLUSION

Majority of the security vulnerabilities are in software and many security weaknesses in software originate in the design stage during the software development process. It is critical to tackle the vulnerabilities in the software design. This research demonstrates a novel approach of selecting appropriate secure design patterns based on the vulnerability anti-pattern model to mitigate common software vulnerabilities in the design. The future work will be focused on two directions. One direction is to track published up-to-date secure patterns and develop a web-based repository of these patterns for the researchers and developers. Another direction is to develop a recommendation tool that applies this approach to identify the anti-patterns of the top eight most common web application vulnerabilities [11] and recommend appropriate secure design patterns.

## REFERENCES

[1] A. K. Alvi and M. Zulkernine, "A Natural Classification Scheme for Software Security Patterns," 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, 2011

[2] X. Xie, C. Ren, Y. Fu, J. Xu and J. Guo, "SQL Injection Detection for Web Applications Based on Elastic Pooling CNN", IEEE Access, vol(7)

[3] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis", Wiley, 1998.

[4] The U.S. Department of Homeland Security (DHS), "Software Assurance," https://www.us-cert.gov/sites/default/files/publications/ infosheet-SoftwareAssurance.pdf, last accessed: March, 2021.

[5] D. Deogun, D. Johnsson and D. Sawano, "Secure by Design," Manning, 2019.

[6] J. Yoder and Jeffrey Barcalow, "Architectural Patterns for Enabling Application Security," Proceedings of Fourth Conference on Patterns Languages of Programs, 1998.

[7] C.Steel, R. Nagappan and R. Lai, Core Security Patterns: Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management, Prentice Hall, 2005.

[8] C. Dougherty, K. Sayre, R. C. Seacord,D. Svoboda and K. Togashi, Secure Design Pattern, Software Engineering Institution, Carnegie-Mellon University, 2009.

[9] E. B. Fernandez, Security Patterns in Practice: Designing Secure Architectures Using Software Patterns, John Wiley & Sons, 2013.

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

[11] O. Ezenwoye, Y. Liu and W. Patten, "Classifying Common Security Vulnerabilities by Software Type," Proceedings of the 32nd International Conference on Software Engineering and Knowledge Engineering, 2020.

[12] T. Nafees, N. Coull, I. Ferguson, and A. Sampson, "Vulnerability anti-patterns: a timeless way to capture poor software practices (vulnerabilities)," In 24th Conference on Pattern Languages of Programs, 2018.

[13] OWASP, "Juice shop application," https://github.com/bkimminich/juice-shop, last accessed: March, 2021.

[14] A. Jafari and A. Rasoolzadegan, Security patterns: A systematic mapping study, Journal of Computer Languages, vol 56, 2020.

[15] F. Buschmann, R. Meunier,H.Rohnert, P. Sommerlad, M.Stal,"A System of Patterns: Pattern-Oriented Software Architecture", Wiley Series in Sotware Design Patterns, 2002.

[16] N. Nahar and K. Sakib, "ACDPR: A Recommendation System for the Creational Design Patterns Using Anti-patterns," IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2016.

[17] Alok Ratnaparkhi, "Repositories", https://github.com/AlokRatnaparkhi,last accessed: March, 2021.

[18] Cross-Site Scripting, https://cwe.mitre.org/data/definitions/79.html,last accessed: May, 2021.