Vulnerability Anti-Patterns: A Timeless Way to Capture Poor Software Practices (Vulnerabilities)

TAYYABA NAFEES, Abertay University, Dundee
NATALIE COULL, IAN FERGUSON, ADAM SAMPSON, Abertay University, Dundee

There is a distinct communication gap between the software engineering and cybersecurity communities when it comes to addressing reoccurring security problems, known as vulnerabilities. Many vulnerabilities are caused by software errors that are created by software developers. Insecure software development practices are common due to a variety of factors, which include inefficiencies within existing knowledge transfer mechanisms based on vulnerability databases (VDBs), software developers perceiving security as an afterthought, and lack of consideration of security as part of the software development lifecycle (SDLC). The resulting communication gap also prevents developers and security experts from successfully sharing essential security knowledge. The cybersecurity community makes their expert knowledge available in forms including vulnerability databases such as CAPEC and CWE, and pattern catalogues such as Security Patterns, Anti-Patterns, and Software Fault Patterns. However, these sources are not effective at providing software developers with an understanding of how malicious hackers can exploit vulnerabilities in the software systems they create. As developers are familiar with pattern-based approaches, this paper proposes the use of Vulnerability Anti-Patterns (VAP) to transfer usable vulnerability knowledge to developers. The primary contribution of this paper is twofold: (1) it proposes a new pattern template – Vulnerability Anti-Pattern – that encapsulates knowledge of existing vulnerabilities to bridge the communication gap between security experts and software developers, and (2) it proposes a catalogue of Vulnerability Anti-Patterns (VAP) based on the most commonly occurring vulnerabilities that software developers can use to learn how malicious hackers can exploit errors in software.

Categories and Subject Descriptors: **Security and privacy~Vulnerability management • Security and privacy~Software security engineering • Software and its engineering~Design patterns**

Additional Key Words and Phrases: Vulnerability, Anti-Pattern, Vulnerability Database (VDB), Vulnerability Anti-Pattern (VAP), Security Pattern, Attack Pattern, Software Development Lifecycle (SDLC)

1 INTRODUCTION

To date, software developers have overlooked security issues throughout the software development lifecycle [1, 2]. The principal reason for this is a lack of understanding about how common errors in software development result in exploitable vulnerabilities in software systems [3]. Existing software design and engineering processes provide little guidance about security, and the communication disconnect between software developers and cybersecurity experts has led to widespread software vulnerabilities [4].

There is no single software engineering technique that guarantees the creation of 100% secure software. Mistakes made by software developers are generally seen as the primary cause of security flaws in software systems. We argue instead that the fault lies with the process: developers lack understanding of how malicious hackers can exploit software flaws, and this understanding is necessary for the creation of secure software.

In our previous work [5], we concluded that software developers do not understand security because their focus is on delivering features, rather than on ensuring security. Accordingly, developers often consider security as something to be added to a system as a bolt-on component in later stages of development. Based on our work that proposed "Caution before Exploitation", we developed an approach based upon the improved use of anti-patterns that encapsulates necessary knowledge about how malicious hackers exploit vulnerabilities.

1.1 Problem Description

The frequency and recurrence of commonly discovered vulnerabilities in databases such as CVE confirms that software developers make the same errors consistently during the development process. Thus, it would be fruitful to study failures, identify the recurring poor software practices and suggest solutions to these problems. This concept is known as a negative pattern or an anti-pattern. Arguably, malicious hackers know a lot more about systems than the developers who created them, indicating that the effectiveness of attackers can be traced back to their extensive knowledge sharing [6]. However, security experts and software developers fail to share this knowledge with each other and, although the problem of frequently recurring software vulnerabilities is very well known, no standard solution has been universally adopted [7]. To solve these problems, it is necessary for experts from both communities to capture and share their knowledge of poor software practices in a form that is suitable for the other party, with clarity, rationale, and context, in a way, which could be applied to a new solution – a pattern.

1.2 Vulnerability Anti-Pattern Objectives

The Vulnerability Anti-Pattern is a hybrid solution, which encapsulates knowledge of vulnerabilities from VDBs and presents this knowledge to developers so that they can understand how poor software practices can be exploited. This increased understanding and awareness of malicious hackers' techniques will contribute to the development of more secure software and aid developers' understanding in the prevention of software vulnerabilities. In essence, Vulnerability Anti-Patterns will:

- **Provide software developers with a conscious understanding of common vulnerabilities using a pattern based approach:** The Vulnerability Anti-Pattern template is based on the anti-pattern approach [8], which will be easily understood by developers. Utilising a pattern-based approach will enable concepts traditionally understood by the security community to be transferred to the software development community, helping developers to identify underlying root causes of vulnerabilities.
- **Bridge the knowledge gap:** The catalogue of Vulnerability Anti-Patterns will provide a common ground to bridge the security awareness gap between both communities' experts – software developers and cybersecurity experts – so they can communicate and work together with confidence in their ability to mitigate vulnerabilities [9, 10].

1.3 Vulnerability: A Commonly Reoccurring Flaw

A Vulnerability may be defined as a commonly reoccurring flaw (also known as an error, mistake, weakness or bug) [11], which generally occurs due to development mistakes, insufficient quality assurance or inadequate security concerns. The work of Martin, Brown et al [12] on the "Top 25 Most Dangerous Software Errors" describes that some software errors occur more frequently in development practices, regardless of whether developers are amateurs or experts, or whether the software is intended for commercial or private use. When coupled with the extensive knowledge sharing in the security community, these frequently occurring errors allow attackers to exploit systems with more efficiency and ease.

1.4 Anti-Patterns: Poor Software Practice

Anti-patterns are derived from design patterns, which capture good practice in software development. Akroyd [13] introduced the idea of anti-patterns as a way of codifying existing practice in the software industry [8]: anti-patterns capture poor practices in software development, along with their causes, solutions and related concerns. The use of anti-patterns allows developers to recognize commonly occurring problems, which may result from a lack of knowledge, insufficient experience in solving a particular type of problem, or applying a correct pattern in the wrong context [14, 15].

In our context, anti-patterns can be used to capture poor software practices that may be exploited by a malicious hacker. Furthermore, they present a framework for the transfer of essential vulnerability knowledge to aid developers in understanding vulnerabilities and identifying appropriate mitigations.

1.5 An Anti-Pattern Perspective for Software Developers

The intended audience for Vulnerability Anti-Patterns are developers who are creating software systems. These developers have other concerns in addition to security: ensuring that industrial standards are met, delivering system features, and meeting time-to-market expectations. It has previously been explained that for software developers, security is the responsibility of "someone else", such as pen-testers and ethical hackers. However, developers have a different set of priorities, and they often misunderstand the importance of security issues [16].

Vulnerability Anti-Patterns aim to provide a solution to this problem through a three-step process integrated into the software development lifecycle: 1) identify the poor software engineering practices which resulted in a particular vulnerability, and understand how the vulnerability can be exploited by an attacker; 2) show the developer how to mitigate this vulnerability; 3) motivate the developer to adopt better security practices in future development, reducing the incidence of future vulnerabilities.

1.6 The Analogy between Anti-Patterns and Vulnerabilities

Our Vulnerability Anti-Pattern template is based on that proposed in Brown et al [14]. However, existing anti-patterns are not intended to capture relationships between poor practices and vulnerabilities, and do not provide mechanisms for capturing cybersecurity domain knowledge.

We argue that existing pattern-based techniques – security patterns [17], software fault patterns [18] and attack patterns [19] – are ineffective at capturing and transferring necessary knowledge of vulnerabilities. Anand, Ryoo et al. (2014) and Hafiz (2011) report that security patterns are harder for developers to use than conventional design patterns. Dimitrov [20] finds that the structure and semantics of SFPs do not adequately

capture all classes of vulnerabilities, and do not align well with existing formal notations used by software engineers. In addition, NIST report that SFPs as used in the CWE database do not appropriately describe the causes or consequences of related vulnerabilities [21]. Faily, Parkin et al [22] evaluated the use of both security patterns and attack patterns within the software development process, and found problems with the identification of specific vulnerabilities in the system, and the complex interactions between security and attack patterns; they report that there is "a dearth of work" evaluating the use of attack patterns by software engineers. The intended audience for security patterns is security experts rather than developers [23, 24], and the need for usable and accessible knowledge about vulnerabilities is highlighted by Van, McGraw [9], Fahl, Harbach et al [25] and Acar, Backes et al [26].

As Vulnerability Anti-Patterns are intended to capture recurring errors that lead to vulnerabilities, we extended the template to include knowledge extracted from vulnerability databases in a form usable by software developers. As an example, the "Use of Potentially Dangerous Function" Anti-Pattern describes the use of functions within a software system that are likely to result in exploitable behaviour, when a safer alternative is available. An instance of this anti-pattern is the use of C's *strcpy* function, which provides no inherent safeguards against incorrect source or target buffer sizes, frequently resulting in faults such as buffer-overflow vulnerabilities [27]. Vulnerabilities resulting from improper use of *strcpy* are so common – as evidenced in the CVE database [28] – that some standards prohibit its use entirely [29]. The "Use of Potentially Dangerous Function" pattern is proposed as a corresponding solution. This Vulnerability Anti-Pattern captures security expert knowledge extracted from these sources in a form that is usable by software developers during the development process.

2 TERMS

2.1 Glossary

Term	Definition
Anti-Pattern	An anti-pattern describes a "general form, the primary causes which led to the general form; symptoms describing how to recognize the general form; the consequences of the general form; and a refactored solution describing how to change the Anti-Pattern into a healthier situation"[14].
Exploit/ Misuse	A "technique to breach the security of a network or information system in violation of security policy" [30].
Vulnerability	A "weakness in a system, application, or network that is subject to exploitation or misuse" [31].
Software Fault Pattern	The Software Fault Patterns (SFP) are a clustering of CWEs into related weakness categories. Each cluster is "factored into formally defined attributes, with sites (footholds), conditions, properties, sources, sinks, etc. This work overcomes the problem of combinations of attributes in CWE" [32].
Penetration Tester	A tester who is "used to test the external perimeter security of a network or facility to find vulnerabilities that an attacker could exploit" [33].

2.2 Acronyms

VDB	Vulnerability Database
VAP	Vulnerability Anti-Pattern
SP	Security Pattern
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
CAPEC	Common Attack Pattern Enumeration and Classification
SDLC	Software Deployment Lifecycle
SFP	Software Fault Pattern

3 OVERVIEW

3.1 The Notion of a Vulnerability Anti-Pattern

A "Vulnerability Anti-Pattern" may be used to describe ineffective practices or poor solutions, which aid the process of identifying and communicating about unsuccessful design intents, and introduces refactored solutions, which suggest safe, alternative procedures. Thus, it can be suggested that these identifiable anti-pattern of poor practices [34] need to be refactored in order to generate safe solutions. The key objective is to stop or mitigate software security flaws from being exploited by attackers. This awareness discrepancy could be attributed to a higher exposure to latent vulnerabilities, which benefits attackers. A possible explanation could be that the prevalent software errors (vulnerabilities) occur, due to established software practices, which have

a negative impact during the software development lifecycle; for example, the use of dangerous function calls. These kinds of practices typically lead to prevalent vulnerabilities.

The scope of anti-patterns has been extended to include vulnerabilities, and introduces another form of anti-patterns termed for the purpose of this research as "Vulnerability Anti-Patterns". The use of anti-patterns for finding and understanding vulnerabilities has not been explored sufficiently, particularly for software developers. This study set out to investigate the use of anti-patterns to provide software developers with the necessary understanding of vulnerabilities. The advantage of adopting Vulnerability Anti-Patterns during the software development lifecycle (SDLC) is that it bridges the knowledge gap between software developers and security experts about commonly occurring software errors. This work has important implications for developing secure training methods.

3.2 A Definition of Vulnerability Anti-Patterns

A Vulnerability Anti-Pattern identifies a problem (i.e. a poor practice that negatively causes a security flaw) and a solution (i.e. a set of refactoring actions that can be carried out to mitigate or stop flaws). The above definition is useful to capture common poor or ineffective practices, which may occur due to the use of design patterns or software development processes in an inappropriate context. Traditionally, it has been argued that software developers have software design and implementation expertise, but lack awareness of security concerns [10, 35, 36].

Table 1 shows the relationship between the anti-pattern and the corresponding pattern in terms of the sources of the information captured by the Vulnerability Anti-Pattern.

Table 1 Relationship between Anti-Pattern and Pattern to describe vulnerability

Anti-Pattern						**Pattern**		
Vulnerability information			**Vulnerability Characteristics**			**Mitigation**		
CWE-ID	**CVE**	**Generally known as**	**Context**	**Lifecycle**	**SFP**	**STRIDE Threat Model**	**SP**	**AP**
CWE-89	CVE-2016-1393, CVE-2015-0161, CVE-2008-5817	SQL Injection	Developing a system with database-driven web sites and save user inputs in a database.	Design Phase	CWE-990: SFP Secondary Cluster: Tainted Input to Command	Spoofing	Intercepting Validator	CAPEC-7, CAPEC-66, CAPEC-108, CAPEC-109, CAPEC-110

3.3 Vulnerability Anti-Pattern (VAP) Template and its Structure

VAPs as presented in this paper follow the standard structure of anti-patterns as shown in Table 2, which includes the following: a general description of the anti-pattern and how it could be exploited, real-world examples sourced from the CVE database and sample vulnerable code, and a risk pattern within SDLC. To mitigate each anti-pattern, a corresponding pattern is proposed, which comprises the following: context, problem, forces, and mitigation. Together, the anti-pattern and pattern can enable developers to recognise the root causes of a vulnerability and help them understand how this vulnerability can be exploited by malicious hackers.

Table 2 Formal VAP Template and its Description

Pattern Main-Division	**Pattern Sub-Division**	**Description**
1. Vulnerability Anti-Pattern General info	**This section describes the general information of the vulnerability**	
	1.1. Anti-Pattern Name	Name of the pattern
	1.2. Also Known as	Other related names that are used to describe this vulnerability
	1.3. Frequently Exposed in SDLC	Mentions the phase of SDLC in which this vulnerability is originally exposed, such as Requirement Specification, Design, Implementation/Coding
	1.4. CWE Mapping: CWE-ID, General Name	Relates this vulnerability to a CWE entry in terms of its ID and name.
	1.5. CVE Example	Reference CVE to give some examples of the related vulnerability
2. Anti-Pattern (Problematic Solution)	**This section explains the poor or ineffective development practices that are captured by this VAP.**	

	2.1. Anti-Pattern Example	Explanation of exploitable scenarios
	2.2. Unbalanced Forces	Unbalanced forces related to meeting requirements, controlling technology changes, and project management.
	2.3. Typical Causes	Causes of this vulnerability as they may be perceived by developers during the SDLC
3. Known Exploitation: Related Anti-Patterns	**This section gives examples of attacks against this particular vulnerability.**	
	3.1. Attack Patterns	Examples of attacks, in reference to CAPEC [19]
Pattern: Solution Against Anti-Pattern		
1. Context	**Relates this pattern to the corresponding Anti-Pattern.**	
2. Problem	**Brief Description about the vulnerability.**	
3. Forces	**This section explains root causes and forensic details of the vulnerability, such as risk patterns and software fault patterns, which formally define attributes and relationships among vulnerabilities.**	
	3.1. Risk patterns and Consequences	Explains the root causes which lead to this vulnerability, including consequences, using the STRIDE model and a description of the vulnerable example.
	3.2. Software Fault Pattern	Links to existing SFPs, sourced from CWE [37]
4. Solution/Mitigation (Refactor the problem)	**This section explains solutions to the vulnerability.**	
	4.1. Solution Steps in SDLC	This portrays all possible solution steps and their mapping within SDLC phases with detailed explanations.
	4.2. Refactored Solution Type	Explains how to refactor this problem by adopting the following patterns. For example, software pattern, technology pattern, process pattern and role pattern.
	4.3. Vulnerable Example (Real-world Patch Example)	An example of how this vulnerability was identified and fixed in a real-world application.
	4.4. Related Patterns	This section demonstrates other related solutions, such as security patterns [38] and language-specific techniques.

3.4 Vulnerability Anti-Patterns (VAP) Catalogue

Table 3 describes our Vulnerability Anti-Patterns (VAP) catalogue.

Table 3 Catalogue of Vulnerability Anti-Patterns

#	Poor Software Practice	Vulnerability Name (sourced from CWE [39])	Vulnerability Anti-Pattern
1	Software fails to correctly escape special elements used in SQL commands	SQL Injection	SQL Injection
2	The software does not perform any authentication for functionality that requires a provable user identity or consumes a significant amount of resources.	Missing Authentication	Missing Authentication for Critical Functions
3	The software does not perform an authorization check when an actor attempts to access a resource or perform an action.	Missing Authorization	Missing Authorization
4	The program copies data to a buffer without checking the size of the input.	Buffer Overflow	Buffer Overflow
5	The program uses a deprecated function call that is prohibited due to known vulnerable behaviour.	Deprecated Function Call	Use of Deprecated Function
6	The program uses a potentially dangerous function that may introduce a vulnerability if used incorrectly.	Use of Potentially Dangerous Function	Use of Potentially Dangerous Function
7	The software performs a calculation that can produce an integer overflow or wraparound.	Integer Overflow	Integer Overflow
8	The software does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.	Incorrect Calculation of Buffer Size	Incorrect Calculation of Buffer Size
9	The software does not properly escape attacker-provided data when generating HTML content.	Cross-Site Scripting (XSS)	Cross-Site Scripting
10	The web application does not sufficiently verify that the source of the request is the same as the target of the request. This enables a	Cross-Site Request Forgery	Cross-Site Request Forgery

	command (triggered from a malicious application) to be sent to a trusted website using the user's browser.		
11	The software uses formatted output functions with a format string controlled by an attacker.	Use of Externally-Controlled Format String	Use of Externally-Controlled Format String
12	The software does not properly escape the special elements used in an operating system command, which may enable execution of arbitrary commands by an attacker.	Shell injection	OS Command Injection

4 VULNERABILITY ANTI-PATTERN EXAMPLES

The first two examples of Vulnerability Anti-Patterns from Table 3 Catalogue of Vulnerability Anti-Patterns are presented below:

1) **SQL Injection Vulnerability**
   I. Anti-Pattern
   II. Pattern (Solution)
2) **Missing Authentication for Critical Functions Vulnerability**
   I. Anti-Pattern
   II. Pattern (Solution)

5 SQL INJECTION VULNERABILITY

5.1 Anti-Pattern

5.1.1 Vulnerability Anti-Pattern General Information

Anti-pattern Name: SQL Injection Anti-Pattern
Also known as: Improper Neutralization of Special Elements used in an SQL Command
Frequently Exposed in SDLC: Design Phase
CWE Mapping:

- CWE-ID: CWE-89
- General Name: SQL Injection

Related CWEs [39]:

CWE-ID	Name
CWE-90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
CWE-564	SQL Injection: Hibernate
CWE-566	Authorization Bypass Through User-Controlled SQL Primary Key
CWE-619	Dangling Database Cursor ('Cursor Injection')

CVE Examples: CVE-2016-1393, CVE-2015-0161, CVE-2008-5817

5.1.2 Anti-Pattern Example

System Specifications that may be Vulnerable to this Example

Software System Type	Any Web-based system with a database
Coding Language	PHP

Example

The scenario described here shows how an SQL injection attack could be carried out, using a user authentication webpage as an example. User details (name and password) are stored in a database, and users are required to enter name and password via input fields (Figure.1).

Figure 1 User login web page

DB Scheme

The web page is linked with an "accounts" table in the database. The "accounts" table stores the values of authenticated users:

Code Description

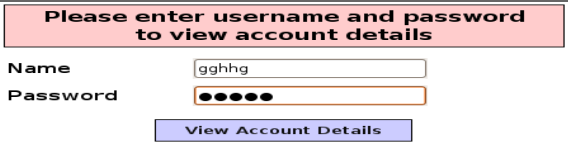
When users select "View Account Details", the following query is executed to retrieve the relevant account details from the database:

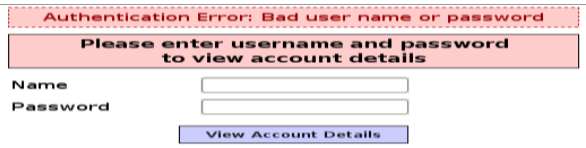
*$lQuery = "SELECT * FROM accounts WHERE username='". $lUsername ."' AND password='" . $lPassword . "'";*

Misuse

This section describes user input values from two different perspectives: software developers' (ideal inputs) and malicious hackers' (exploitation inputs), and the corresponding outputs.

Ideal Case

Input	Description	Screen shot
Ideal input Name: *gghhg* Password: *1234*	Alphanumeric characters in name and password fields.	

Ideal output	If the user information is valid, the user's account details are displayed; if not, an error message (Figure 3) is displayed.	 Figure 3 Output screen

Exploitation Case

Input	Description	Screen shot
Exploitation input Name: *' OR 1=1 #*	Malicious input is interpreted as SQL commands rather than username data.	
Exploitation output	The SQL statement constructed by the program returns all records from the database table, rather than just the intended record.	

Unbalanced Forces: Management of performance and complexity

Unbalanced Forces	Attack Description	Example(C#)
Lack of input validation	The program is not validating the input data and the SQL query is constructed dynamically based on external input data. This allows the attacker to change the semantics of the SQL query.	1. `Sql.Open();` 2. `String sqlstring= " SELECT ccnum" + " FROM cust WHERE id=" +Id;` 3. `SqlCommand cmd= new SqlCommand (sqlstring,sql);` 4. `Ccnum=(string)cmd.ExecuteScalar();`
Lack of intercepting filter	An attacker can easily construct malicious input to inject an SQL query and get access to critical information stored in the database, such as passwords.	1. `string userName = ctx.getAuthenticatedUserName();` 2. `string query = "SELECT * FROM items WHERE owner = '" + userName + "' AND itemname = '" + ItemName.Text + "'";` 3. `sda = new SqlDataAdapter(query, conn);` 4. `DataTable dt = new DataTable();` 5. `sda.Fill(dt);`

Typical Causes [14]

- **Lack of experience:** This is especially common in software developers who lack understanding of secure coding practices.
- **Lack of Input Validation:** The system does not validate external input data coming from all other sources.

5.1.3 Known Exploitation (Attack Patterns [19])

CAPEC-7, CAPEC-66, CAPEC-108, CAPEC-109, CAPEC-110

5.2 Pattern

5.2.1 Context

The context, in which the SQL Injection Pattern can be applied is already explained in SQL Injection Anti-Pattern.

5.2.2 Problem

The system does not correctly escape special characters used in the construction of an SQL query that may alter the meaning of the query passed to the database. Attackers can easily inject their own SQL code into the executed query, allowing a wide variety of actions. For example, when SQL queries are used in security contexts such as authentication, attackers may modify the logic of those queries to gain unauthorized access by adjusting authentication rules, or deleting or updating database records. SQL injection attacks may target SQL queries being constructed by application code directly, or queries performed by client- or server-side stored procedures. Several well-known attacks involve compromising a system by injecting SQL queries containing malicious code or database content.

5.2.3 Forces

Risk patterns	Consequences	Context Description
• No sanitisation of input data when building the SQL statement. • No separation between data and code while building the SQL statement. • Not validating SQL statement parameters for boundary checking, data size and format. • No internal validation check on database queries and transactions.	Spoofing	A system with a database that takes external user inputs.

SFP Secondary Cluster [40]: Tainted Input to Command

- **Mitigation Pattern: Sanitize External Input**
  Developer must use the input validation strategy to sanitize all external user provided inputs during the construction of SQL queries and must follow the corresponding language input validation approaches.

5.2.4 Mitigation

Solution Steps in SDLC [41]

SDLC phase	Solution
Design Phase	• Secure the UML design to validate data supplied by clients for malicious code or malformed content. The data could be form-based, queries, or XML content. • Select a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Example: using persistence layers such as Hibernate or Enterprise Java Beans.
Implementation Phase	• Verify all external inputs before use. Use a pluggable filters approach and apply declarative filters based on URL. Restrict filter tasks to do pre-processing of all requests and provide validation. Perform sever side validation, because client side validation is not secure and open to spoofing. Renegotiate trust between users after a specific interval of time. Log queries performed and identify irregular behaviour. • Dynamic query construction is generally considered as an insecure development practice, but in some contexts, it may be unavoidable. In these cases, always perform careful sanitization of query arguments with correct escaping of special characters within those arguments.

Refactored Solution Name: Sanitizing Input data

Refactored Solution Type [17] :

- **Technology Pattern:** The J2SE 1.4 regular expression package provides classes that allow developers to perform regular expression matches. These classes can be used for data validation prior to its use in query generation.
- **Software Pattern:** Intercepting filter pattern (CJP2) for data validation.
- **Process Pattern:** Many programming languages provide dynamic facilities to intercept, scrub and validate data prior to its use during the construction of queries; for example, the Oracle DBMS_ASSERT package.

Vulnerable Examples [37]

Vulnerable Example Information		
Product Versions	**Name**	**CVE-ID**

Cisco Cloud Network Automation Provisioner Releases 1.0 and 1.1.	Cisco Cloud Network Automation Provisioner SQL Injection Vulnerability	CVE-2016-1393
Vulnerable code/ Attack Method	An attacker could exploit this vulnerability by sending crafted URLs that include SQL statements to a targeted system. A successful exploit could allow the attacker to modify or delete entries in some database tables.	
Misuse of the vulnerable code	This vulnerability occurred due to a failure to validate user input in constructing SQL queries, in the web framework of Cisco Cloud Network Automation Provisioner. CNAP could allow an authenticated, remote attacker to affect the integrity of an affected system by executing arbitrary SQL queries.	
Patch (Solution)	Cisco has not released software updates that mitigate this vulnerability.	

Related Patterns

- Intercepting Validator
- Message Interceptor Gateway
- J2EE Applications Solution
- Message Interceptor
- Interceptor (POSA)

6 MISSING AUTHENTICATION FOR CRITICAL FUNCTIONS VULNERABILITY

6.1 Anti-Pattern

6.1.1 Anti-pattern General Information

Anti-Pattern Name: Missing Authentication Anti-Pattern
Also known as: Missing Authentication
Frequently Exposed in SDLC: Design Phase
CWE Mapping:

- CWE-ID: CWE-306
- General Name: Missing Authentication for Critical Functions

Related CWEs [39]:

CWE-ID	Name
CWE-302	Authentication Bypass by Assumed-Immutable Data
CWE-307	Improper Restriction of Excessive Authentication Attempts

CVE Example: CVE-2002-1810, CVE-2008-6827, CVE-2004-0213

6.1.2 Anti-Pattern Example

System Specification that may be Vulnerable to this Example

Software System Type	Web Application
Coding Language	Any scripting language capable of interacting with authentication functionality.

Example

The presented scenario indicates how a system could be exploited by malicious hackers due to lack of authentication, thereby granting access to the critical parts of the system to illegitimate users. As shown in Figure.1, the login page has two input fields: "Login" and "Password".

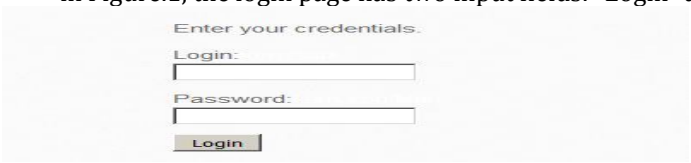


Figure 1 Login web page

Misuse

This section examines login (critical) function inputs from two different perspectives – software developers' (ideal inputs) and malicious hackers' (exploitation inputs) – and the corresponding outputs.

Ideal Case

Input	Description	Screen Shot
Ideal input	According to the developer's ideal case, the user requests to log_in function (critical function) of the system by providing valid username and password in input fields "Login" and "Password".	
Ideal output	As shown in Figure 2, according to the developer's ideal output, if the user inputs valid information, the log_in function successfully permits access to the system.	 Figure 2 Login function output message

Exploitation Case

Input	Description	Screen Shot
Exploitation input	However, during the implementation of the log_in function, the developer forgets to implement user authentication, allowing the attacker to input any random value in the input fields "Login" and "Password" instead of valid user credentials.	
Exploitation output	Web_View of login function shown in Figure 3: after passing the invalid credentials to the log_in function, because this function has no authentication mechanism to validate the user information, the attacker can get access to the system as a legitimate user.	 Figure 3 Login function

Unbalanced forces: Management of Performance, Complexity and IT resources

Unbalanced forces	Attack Description	Example
Lack of authentication enforcement	Attacker can simply bypass authentication and misuse system resources.	A security-critical system is implemented for testing purposes without user authentication, and is later used within a production environment without the lack of authentication implementation being identified during testing.

Typical Causes [14]

- **Lack of Architecture, or Non–Architecture Driven Development:** This is especially prevalent with transient development teams, which may lack awareness of the overall architecture of a system and the rights of the actors within that system.
- **Limited Intervention:** In iterative projects that extend a legacy system, developers may prefer to add little pieces of functionality to existing working code, rather than revising the system design for more effective allocation of rights for the new actors.
- **Lack of Authorization Architecture Enforcement:** When the implementers are not aware of the need for authorization enforcement, it does not matter how effective the system design is in terms of authentication; developers may implement their own authentication approaches, in ways that are not consistent with the system design.
- **Lack of configuration:** The system lacks configuration management or compliance with process management policies.

6.1.3 *Known Exploitation (Attack Patterns [19])*

CAPEC-225, CAPEC-12, CAPEC-36, CAPEC-40, CAPEC-62

6.2 Pattern

6.2.1 Context

The context, in which the Missing Authentication for Critical Functions Pattern can be applied is already explained in the Missing Authentication for Critical Functions Anti-Pattern.

6.2.2 Problem

It is critical to prevent a system or its resources from illegitimate access. A malicious attacker may try to impersonate as a legitimate user to gain access to system critical resources. This may be particularly serious if the user impersonated has high level of privileges. This flaw originates mostly during the design phase, when the software developer neglects to require user authentication before giving access to valuable resources, and does not make use of a centralized authentication system, or when the developer does not consider the need to verify user identity from all potential communication channels.

6.2.3 Forces

Risk Patterns	Consequences	Context Description
• Lack of a centralized user authentication system. • Lack of identity verification when users access the system or resources from some communication channels. • Lack of use of authentication facilities provided by an implementation framework or operating system. • Lack of separation between authentication tasks and authorization tasks.	Spoofing	A software system that needs to protect valuable resources for its users or their institutions.

SFP Secondary Cluster [40]: Missing Authentication

- **Mitigation Pattern: Enforce Actor Management and Authentication**
  Developer must restrict access of the system only to valid users with preassigned privileges. The authentication mechanism must be properly enforced throughout the development of the system.
- **Mitigation Pattern: Missing Endpoint Channels Authentication**
  There may be multiple entry points into the system, such as client and administrator interfaces; each of these must assure authentication of the user.

6.2.4 Mitigation

Solution Steps in SDLC [39]

SDLC Phase	Solution
Requirement Specification Phase	Secure UML design includes misuse use-cases, possible attack methods and appropriate security patterns.
Design Phase	• Divide the software authentication into guest, normal, privileged, and administrative parts. Identify and verify which of these parts require a verified user identity, and implement a centralized authentication capability. • Avoid executing custom authentication routines. Where possible, use authentication capabilities as provided by the implementation framework or operating system. • Provide a well-defined separation between authentication and authorization tasks – in online systems, authentication and authorization are usually not well separated. If the system lacks a centralized authentication system, then verification must be explicitly applied for all interactions with system resources.

Refactored Solution Name: Safeguard legitimate users and critical resources
Refactored Solution Type [17]:

- **Technology pattern: Java Authentication Enforces:** In J2EE applications, the Java Authentication Enforces pattern provides a consistent and structured way to handle authentication and verification of requests across and within web-tier components, and supports a Model-View-Controller (MVC) architecture without code duplication.
- **Process pattern:** Ensure that all user interactions with the system pass through a single point of access, and apply a standard authentication protocol (such as CHAP) to verify the identity of the user. The use of protocols may be simple or complex, depending on the needs of the application and domain.

Vulnerable Example [37]

Vulnerable Example General Information		
Product Versions	**Name**	**CVE-ID**
Drupal 6.x before version 6.2.	Drupal-Menu_System-Security-Bypass	CVE-2008-1729
Vulnerable code/ Attack Method	Attackers can exploit the Menu System in versions of Drupal 6 prior to 6.2, which has incorrect menu settings. It allows remote attackers to edit the profile pages of arbitrary users and obtain sensitive information from the tracker, and allows remote authenticated users with administration page view access to edit content types.	
Misuse of the vulnerable code	The Drupal Menu System could allow a remote attacker to bypass security restrictions, caused by improper restrictions on certain pages. An attacker could exploit this vulnerability to edit user profile pages, edit content types on administrative pages, and obtain tracker and blog page information when access content permission is disabled	
Patch (Solution)	The vendor has released updates that address this vulnerability. Refer to DRUPAL-SA-2008-026 for the patch.	

Related Patterns

- Authentication Enforcer
- Single Access Point
- Policy Enforcement Point
- J2EE Applications Solution
    - Container Authentication strategy
    - Authentication Provider-Based strategy (using third party product)
    - JAAS Login Module strategy

7 CONCLUSION

To mitigate vulnerabilities, we anticipate that Vulnerability Anti-Patterns will encourage developers to maintain a deeper understanding and conscious understanding of vulnerabilities in their future development practices. We propose a template for anti-patterns, and using this template we are able to produce a catalogue of Vulnerability Anti-Patterns against 12 vulnerabilities (Table 3 Catalogue of Vulnerability Anti-Patterns), chosen from the OWASP list of "Top 25 Most Dangerous Software Errors". These proposed VAPs are currently being evaluated to measure their effectiveness in providing real-world software developers with awareness of common software vulnerabilities.

8 REFERENCES

1. Mansourov N., Campara D.: System assurance: beyond detecting vulnerabilities. Elsevier (2010).

2. Shiralkar T., Grove B.: Guidelines for secure coding. (2009).

3. Morgan S.: Is poor software development the biggest cyber threat?. http://www.csoonline.com/article/2978858/application-security/is-poor-software-development-the-biggest-cyber-threat.html (2016).

4. Hans K.: Cutting edge practices for secure software engineering. 4, 403-408 (2010).

5. Nafees T., Coull N., Ferguson R.I., Sampson A.: Idea-caution before exploitation: The use of cybersecurity domain knowledge to educate software engineers against software vulnerabilities. , 133-142 (2017).

6. Mansourov N., Campara D.: Why hackers know more about our systems. , 1-21 (2011).

7. Aslam T., Krsul I., Spafford E.H.: Use of a taxonomy of security faults. (1996).

8. Akroyd M.: Anti patterns session notes. (1996).

9. Van Wyk K. R., McGraw G.: Bridging the gap between software development and information security. 3, 75-79 (2005).

10. McGraw G.: Software assurance for security. 32, 103-105 (1999).

11. Ilyin Y.: **Can we beat software vulnerabilities?**. https://business.kaspersky.com/can-we-beat-software-vulnerabilities/2425 (2015).

12. Martin B., Brown M., Paller A., Kirby D., Christey S.: 2011 CWE/SANS top 25 most dangerous software errors. 7515 (2011).

13. Akroyd M.: AntiPatterns: Vaccinations against object misuse. (1996).

14. Brown W.H., Malveau R.C., McCormick H.W., Mowbray T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc. (1998).

15. Foote B., Yoder J.: Big ball of mud. 4, 654-692 (1997).

16. McGraw G.: Software security: building security in. Addison-Wesley Professional (2006).

17. Steel C., Nagappan R.: Core Security Patterns: Best Practices and Strategies for J2EE", Web Services, and Identity Management. Pearson Education India (2006).

18. Mansourov D. N.: Software fault patterns: Towards formal compliance points for CWE. (2011).

19. MITRE Corporation: Common attack pattern enumeration and classification. http://capec.mitre.org/index.html (2014).

20. Dimitrov V.: Toward formalization of software security issues. (2016).

21. Black P. E.: SARD: A software assurance reference dataset. (2017).

22. Faily S., Parkin S., Lyle J.: Evaluating the implications of attack and security patterns with premortems. In: Cyberpatterns, pp. 199-209. Springer (2014).

23. Bunke M.: Software-security patterns: Degree of maturity. , 42 (2015).

24. Fernandez-Buglioni E.: Security patterns in practice: designing secure architectures using software patterns. John Wiley & Sons (2013).

25. Fahl S., Harbach M., Perl H., Koetter M., Smith M.: Rethinking SSL development in an appified world. , 49-60 (2013).

26. Acar Y., Backes M., Fahl S., Kim D., Mazurek M.L., Stransky C.: You get where you're looking for: The impact of information sources on code security. , 289-305 (2016).

27. Howard M.: Security development lifecycle (SDL) banned function calls. 2011 (2011).

28. MITRE Corporation: **Cve-2016-8820**. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8820 (2016) 2017.

29. OWASP: **Buffer overflows**. https://www.owasp.org/index.php/Buffer_Overflows (2015).

30. Instruction C.: 4009,"National information assurance glossary," committee on national security systems, may 2003. 4009 (2003).

31. Kissel R.: NIST IR 7298 revision 2: Glossary of key information security terms. 7, 2013 (2013).

32. Mansourov D. N.: Software fault patterns: Towards formal compliance points for CWE. (2011).

33. CERT: **Common cyber security language - ICS-CERT - US-CERT**. (2015).

34. Julisch K.: Understanding and overcoming cyber security anti-patterns. 57, 2206-2211 (2013).

35. Viega J., McGraw G.: Building Secure Software: How to Avoid Security Problems the Right Way, Portable Documents. Pearson Education (2001).

36. McGraw G.: Software security. 36, 662-665 (2012).

37. MITRE Corporation: **Common vulnerabilities and exposures (CVE)**. https://cve.mitre.org/ (2015).

38. Schumacher M., Fernandez-Buglioni E., Hybertson D., Buschmann F., Sommerlad P.: Security patterns. In: Security patterns: Integrating security and systems engineering. John Wiley & Sons (2013).

39. MITRE Corporation: **Common weakness enumeration**. http://cwe.mitre.org/ (2015).

40. MITRE Corporation: **CWE-888: Software fault pattern (SFP) clusters** (2015). Accessed 08 2014.

41. MITRE Corporation: **CWE-89: Improper neutralization of special elements used in an SQL command ('SQL injection')**. (2012).