

An Attack Graph Generation Framework based on Attack Patterns

*Project report submitted in partial fulfillment of the requirements
for the degree of*
Master of Computer Application

by

Tuhinsuvra Sarkar

Class Roll No.: 002210503003

Examination Roll No.: MCA244003

Registration No.: 163595 of 2022-2023

under the supervision of

Dr. Mridul Sankar Barik

Department of Computer Science and Engineering
Faculty of Engineering and Technology
Jadavpur University
2024

Faculty of Engineering and Technology Jadavpur University

Certificate of Recommendation

To Whom It May Concern

I hereby recommend that the project report entitled “**An Attack Graph Generation Framework based on Attack Patterns**” has been carried out by **Tuhinsuvra Sarkar** (Registration No.: 163595 of 2022-2023, Class Roll No.: 002210503003, Examination Roll No.: MCA244003), under my guidance and supervision and be accepted in partial fulfilment of the requirements for the degree of Master of Computer Application in the Department of Computer Science & Engineering, Jadavpur University, during the academic year 2023-2024.

Dr. Mridul Sankar Barik
Assistant Professor
Dept. of Comp. Sc. & Engg.
Jadavpur University
(**Project Supervisor**)

Prof. Nandini Mukhopadhyay
Head of the Department
Dept. of Comp. Sc. & Engg.
Jadavpur University

Prof. Dipak Laha
Dean,
Faculty of Engineering & Technology
Jadavpur University

Faculty of Engineering and Technology

Jadavpur University

Certificate of Approval

This is to certify that the project report entitled "**An Attack Graph Generation Framework based on Attack Patterns**" is a bonafide record of work carried out by **Tuhinsuvra Sarkar**, in fulfillment of the requirements for the award of the degree of Master of Computer Application in the Department of Computer Science and Engineering, Jadavpur University. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the project report only for the purpose for which it has been submitted.

Signature of Examiner 1

Date:

Signature of Examiner 2

Date:

Faculty of Engineering and Technology Jadavpur University

Declaration of Compliance with Academic Ethics

I hereby declare that this project report entitled "**An Attack Graph Generation Framework based on Attack Patterns**" is an outcome of my genuine and independent work, as part of my Master of Computer Application program. All information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, all materials and results that are not original to this work have been fully cited and referenced.

Name: Tuhinsuvra Sarkar

Registration No.: 163595 of 2022-2023

Examination Roll No.: MCA244003

Thesis Title: An Attack Graph Generation Framework based on Attack Patterns

Signature with Date

Acknowledgement

With my most sincere and gratitude, I would like to thank Dr. Mridul Sankar Barik, Department of Computer Science & Engineering, my supervisor, for his overwhelming support throughout the duration of the project. His motivation always gave me the required inputs and momentum to continue with my work, without which the project work would not have taken its current shape. His valuable suggestions and numerous discussions have always inspired new ways of thinking. I feel deeply honored that I got this opportunity to work under him.

I would like to express my sincere thanks to all my teachers for providing sound knowledge base and cooperation. I would like to thank all the faculty members of the Department of Computer Science & Engineering of Jadavpur University for their continuous support.

Last, but not the least, I would like to thank my batch mates for staying by my side when I needed them.

Tuhinsuvra Sarkar

Class Roll No.: 002210503003

Examination Roll No.: MCA244003

Department of Computer Science & Engineering

Jadavpur University

Contents

1	Introduction	1
1.1	Background	1
1.2	Research Statement	2
1.3	Contribution of the project	2
1.4	Outline of the project	3
2	Attack Graphs	4
2.1	Overview	4
2.2	Generation of Attack Graph	4
3	Attack Patterns	8
3.1	Threat Model and Attack Model	8
3.2	CAPEC	9
3.3	CWE	10
3.3.1	Pillars	10
3.3.2	Classes	11
3.3.3	Base	12
3.3.4	Variant	13
3.3.5	Categories:	14
3.4	CVE	14
3.5	Attack Pattern	18
3.5.1	Category Attack Pattern	19
3.5.2	Meta Attack Pattern	19
3.5.3	Standard Attack Pattern	21
3.5.4	Details Attack Pattern	24
3.6	CAPEC vs. ATT&CK	26
3.6.1	What is CAPEC	26
3.6.2	What is ATT&CK	27
4	Proposed Methodology	28
4.1	Methodology	28
4.2	Algorithm	29

5	Experiments	35
5.1	Objective	35
5.2	Hypothesis	35
5.3	Experimental Setup	36
5.4	Example	36
5.5	Limitations	39
6	Conclusion and Future Work	41
6.1	Conclusion	41
6.2	Future Work	42

List of Figures

2.1	Example Network Configuration	5
2.2	List of Vulnerabilities of the Example Network Configuration	5
2.3	Attack Graph of the Example Network Configuration	6
3.1	Category attack patterns	19
3.2	Meta Attack Pattern	21
3.3	Working Principles of SQL Injection Attack	23
3.4	Standard Attack Pattern	24
3.5	Details of the Attack Pattern of Figure 3.4	26
4.1	Overview of the Proposed Methodology	29
5.1	System Model	37
5.2	Detected Attack Patterns Between H1 to H4	38
5.3	Detected Attack Patterns Between H1 to H2 and H2 to H4	38
5.4	Detected attack patterns Between H4 to H3	39
5.5	System Model	39

Abstract

Attack graphs have become an essential tool for identifying multi-stage, multi-host attacks within organizational networks. As networks grow more extensive and complex, even experienced security analysts struggle to pinpoint how attackers exploit vulnerabilities across multiple hosts to compromise critical resources. Traditional methods of generating attack graphs face scalability challenges and often require frequent adaptations due to network configuration changes. Our project addresses these issues by introducing a novel attack graph generation framework based on attack patterns.

We leverage the power of graph databases to manage large, interconnected datasets efficiently. Our framework uses Common Attack Pattern Enumeration and Classification (CAPEC) to map vulnerabilities to corresponding attack patterns, enabling a comprehensive representation of potential attack paths. By utilizing the Neo4j graph database, we enhance the scalability and practicality of attack graph analysis.

This project details the development of our framework, including an efficient algorithm for identifying all possible attack paths within a network. The approach takes into account network configurations, known vulnerabilities, and potential attack patterns, providing a nuanced understanding of a network's security posture. Our system generates attack graphs dynamically and facilitates interactive analysis, significantly improving the detection and mitigation of security threats.

Experimental results demonstrate the effectiveness of our approach in real-world scenarios, highlighting its potential to offer valuable insights for network administrators, security analysts, and researchers. By regularly updating the attack graph based on the latest vulnerability data, our framework ensures timely detection of changes that could pose significant security risks. This work aims to provide a robust and scalable solution for network security analysis, enhancing the ability to monitor and defend against complex, evolving threats.

Chapter 1

Introduction

Attack graphs have emerged as a valuable tool for identifying multi-stage, multi-host attacks within organizational networks. Without such tools, even experienced security analysts struggle to uncover how attackers exploit vulnerabilities across hosts to compromise crucial resources, especially as networks grow larger and more complex. Having an attack graph that maps out all potential attack paths is vital for security administrators to grasp the breadth of threats and implement appropriate countermeasures.

Previous research has attempted to address these challenges, often using proprietary algorithms for attack graph analysis rather than standard graph-based techniques. However, these approaches may require frequent adaptation due to network or security configuration changes. Thus, there's a demand for solutions that facilitate interactive attack graph analysis.

In recent years, graph databases have regained popularity for managing large graph data sets efficiently, offering superior performance when dealing with interconnected data such as social networks, web structures, or biological relationships. Transitioning from relational to graph databases can significantly enhance the scalability and practicality of analyzing attack graphs and similar complex data structures.

In our project, we introduce a graph framework designed to generate attack graphs based on the network state. Additionally, we demonstrate how graph queries can be leveraged to create attack graphs and conduct common analysis tasks on these generated graphs. Our implementation utilizes the widely used Neo4j graph database to realize this proposed model effectively.

1.1 Background

Early research focused on efficient generation of attack graphs, with initial approaches using symbolic model checking to analyze host configurations and vulnerabilities. However, these methods faced scalability issues due to the exponential increase in nodes representing system states.

To address scalability, automated approaches, such as the TVA approach, represented different network configurations rather than individual system states in each node of the attack graph. This approach assumed the monotonicity property of attacks and used a graph search algorithm to chain vulnerabilities and find attack paths.

Tools like NetSPA and MULVAL have been developed to generate attack graphs from firewall

rule sets and network vulnerability scans, respectively. These tools analyze the attack graph to provide prioritized recommendations and capture the interaction among various network components.

Our current work is motivated by the need for efficient storage and retrieval of attack graphs for easier and on-the-fly subsequent analysis tasks. We propose an attack graph generation framework based on attack patterns, an abstract description of common approaches attackers take to exploit analogous vulnerabilities. This framework maps vulnerabilities to corresponding attack patterns and uses a database for generation, storage, and query of attack graphs.

This approach enhances existing analysis techniques and opens up opportunities for new ones. It focuses on properties of different attack paths in the attack graph, aiding in intrusion alert correlation, network hardening solutions, and security metrics based on attack path characteristics. Despite the advancements, efficient storage and retrieval of attack graphs remain a challenge, which our framework aims to address.

1.2 Research Statement

Modern computer networks are in a state of constant evolution due to frequent configuration changes, diverse application portfolios, and various network devices. This leads to a continuously expanding threat landscape and a complex network with a varying subset of configurations and vulnerabilities that an adversary could exploit. The attack graph of the network, which represents potential attack paths, is subject to constant change due to the discovery of new vulnerabilities, misconfigurations, and changes in access control policies.

The focus of this project is to introduce the concept of generating an attack graph based on attack patterns. We propose a novel approach to model the network’s security posture by mapping vulnerabilities to corresponding attack patterns. This approach allows us to generate a more comprehensive and meaningful representation of potential attack paths in the network.

We will present an efficient algorithm to find all possible attack paths in a network. This algorithm takes into account the network’s configuration, known vulnerabilities, and potential attack patterns to identify paths that an adversary could potentially exploit.

In this project, we will present the algorithms for our proposed framework and analyze their performance. We will demonstrate how our approach can provide a more nuanced understanding of a network’s security posture, enabling more effective monitoring and mitigation strategies. By regularly updating the attack graph based on the latest vulnerability and attack pattern data, we can ensure timely detection of changes in the network that could pose a significant risk to its security.

This work aims to provide a robust and scalable solution for network security analysis, offering valuable insights for network administrators, security analysts, and researchers in the field.

1.3 Contribution of the project

In this project, we introduce the concept of an attack graph and its associated attack paths, providing a comprehensive definition and illustrative examples. We delve into the specifics of Common Attack Pattern Enumeration and Classification (CAPEC) attack patterns, detailing their types with diagrams and examples for clarity. We also define Common Weakness Enumeration (CWE) and their types, and discuss the relationship between CAPEC, CWE, and Common Vulnerabilities

and Exposures (CVE). A comparison between CAPEC and ATTACK is presented to highlight their differences.

After establishing these foundational concepts, we discuss our approach to generating an attack path from a given network schema. Initially, the schema is provided through a CSV file. Our system utilizes this schema to generate a comprehensive network graph. We then perform a Depth-First Search (DFS) to identify all possible attack paths within the network.

We present the output in a graph format, labeling each path with the name of the corresponding attack pattern. This visual representation aids in identifying potential entry points for an attacker within the system.

This project presents algorithms for our proposed framework and analyzes their performance. We demonstrate how our approach can provide a more nuanced understanding of a network's security posture, enabling more effective monitoring and mitigation strategies. By regularly updating the attack graph based on the latest vulnerability and attack pattern data, we can ensure timely detection of changes in the network that could pose a significant risk to its security.

This work aims to provide a robust and scalable solution for network security analysis, offering valuable insights for network administrators, security analysts, and researchers in the field.

1.4 Outline of the project

The structure of this project begins with an abstract, followed by an introduction in Chapter 1, which includes the research statement and contribution of the project. Chapter 2 briefly covers related materials. Chapter 3 presents attack patterns, while Chapters 4 and 5 cover the methodology and algorithm of the model. Chapter 6 presents the proposed work of this project, which is attack graph generation based on attack patterns, and includes details on the experiments conducted. Lastly, Chapter 7 presents the conclusion of the work completed in this project.

Chapter 2

Attack Graphs

2.1 Overview

An attack graph is a graphical representation that depicts the possible attack routes an attacker may use to infiltrate a system or network. It models the interactions between various system components, including vulnerabilities, assets, and possible attack vectors. Automated tools that analyze system settings, vulnerability information, and other pertinent data sources are often used to generate these graphs.

Attack graphs are useful for enumerating multi-stage, multi-host attacks in organizational networks. They show all possible multi-stage, multi-host attack paths, which is crucial for a security administrator to understand the diverse nature of threats and decide on appropriate countermeasures.

These graphs are used for measuring network security and aid many security solutions. For instance, intrusion alert correlation techniques try to detect multi-step intrusions by correlating isolated alerts of individual attack steps. Network hardening solutions use attack paths enumerated in an attack graph to identify vulnerabilities that should be removed such that none of the attack paths leading to a given critical resource can be realized.

2.2 Generation of Attack Graph

Graphs are a natural choice for formalizing automated network security analysis techniques [1]. Attack graphs originated from privilege graphs which are used to describe network system vulnerabilities. In a privilege graph, each node represents a set of privileges owned by a user or a set of users, and each edge represents a vulnerability. In an attack tree, each path to a leaf node represents a sequence of attacks that an attacker can use to reach the goal state from the initial state. An attack graph is essentially a consolidated representation of an attack tree, where some or all common nodes across different attack paths are merged.

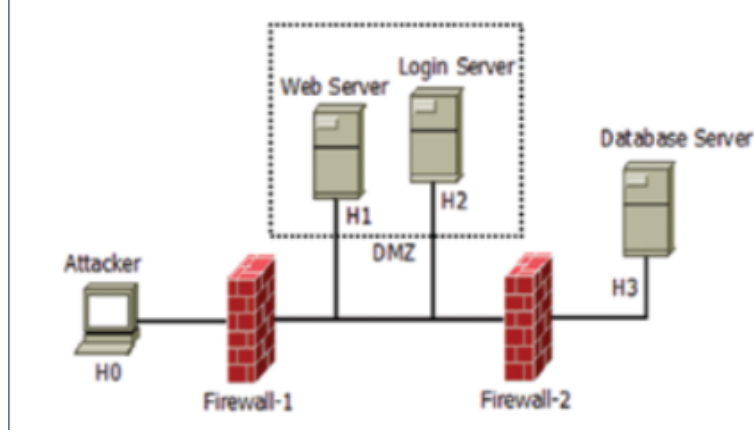


Figure 2.1: Example Network Configuration

Consider the network configuration shown in Figure 2.1. In this configuration, Firewall-1 controls traffic between the external and internal networks. The attacker is assumed to be located on host H0 in the external network. In the DMZ of the internal network, a web server runs on host H1, and a login server (via SSH) runs on host H2. The web service requires access to a backend database server running on host H3. Firewall-1 allows HTTP traffic to the web server and SSH traffic to the login server, while blocking all other traffic. Firewall-2 only allows access to the database server from the web server.

Host H1 runs a vulnerable version of the Apache web server, which has a vulnerability (CVE-2006-3747) that allows a remote attacker to gain user privileges on the web server. The SSH service on H2 has a vulnerability (CVE-2002-0640) that allows remote attackers to gain user privileges. The database server on H3 is a Linux box running MySQL, which has a remotely exploitable vulnerability (CVE-2009-2446) that allows attackers to gain user privileges. Additionally, the Linux kernel on host H3 has a vulnerability (CVE-2004-0495) that allows local users to gain root privileges. The attacker's objective is to gain root privileges on the database server.

For notational convenience, short symbols are used for different vulnerabilities, as shown in Table 2.2.

Table 1. Short symbols used for different vulnerabilities

Software	Vulnerability (CVE-ID)	Short symbol
Apache Web Server v1.3,	CVE-2006-3747	V1
OpenSSH v2.3.1-v3.3	CVE-2002-0640	V2
MySQL v4.0, v5.0	CVE-2009-2446	V3
Linux Kernel v2.4, v2.6	CVE-2004-0495	V4

Figure 2.2: List of Vulnerabilities of the Example Network Configuration

Early attack graph generation methods faced scalability issues due to the exponential state-space representation of state enumeration or scenario graphs. The monotonicity assumption states that the preconditions of an attack are never invalidated by another successful attack. This reduces analysis complexity from exponential to polynomial, leading to the creation of the exploit dependency graph, which has a worst-case quadratic relationship to the number of exploits.

The Topological Vulnerability Analysis (TVA) approach uses this principle, focusing on modeling attacker exploits and finding sequences of attacks from the initial state to critical network assets. TVA requires a comprehensive knowledge base of vulnerabilities and attack techniques. The TVA attack graph generation algorithm, originally with $\mathcal{O}(n^6)$ complexity, was improved to $\mathcal{O}(n^3)$, where n is the number of network hosts.

The TVA model employs exploit nodes and security condition nodes. Exploit nodes represent attacks and are defined by pre- and post-conditions. Security condition nodes represent either the pre- or post-conditions. Directed edges indicate these conditions, simplifying the complexity by modeling exploits on hosts rather than the hosts themselves.

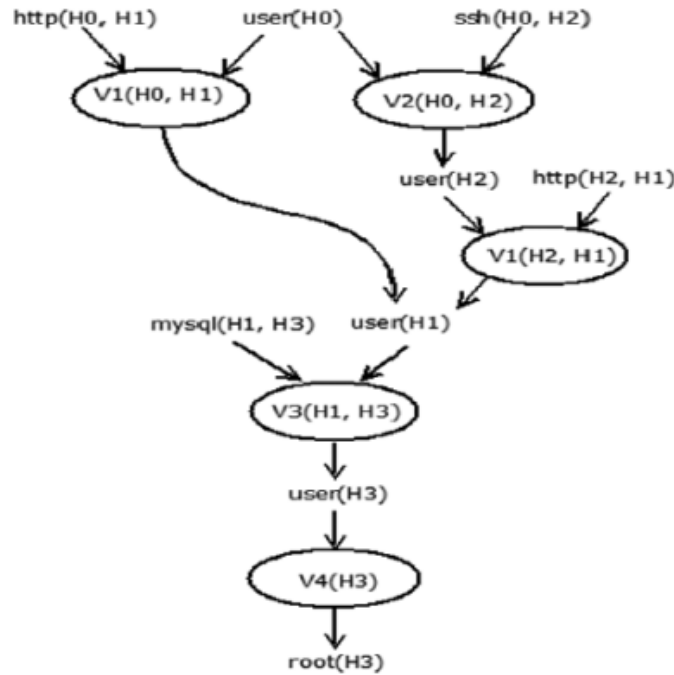


Figure 2.3: Attack Graph of the Example Network Configuration

For example, network condition `http (H0, H1)` means accessibility of web service on host H1 from host H0. Attacker capability `user(H0)` means, attacker has user privilege on host H0. Directed edges in and out of exploit nodes identify pre and post-conditions respectively of an attack. For example, exploitation of MySQL vulnerability CVE-2009-2446 on host H3 from host H1 i.e. `V3(H1, H3)` requires pre-conditions `user(H1)` and `mysql(H1, H3)` and generates a post-condition `user(H3)`. Exploit dependency attack graph elegantly enumerates different attack paths leading to a critical

resource. In this example, there are two attack paths leading to attacker gaining root privilege on H3. They are $V1(H0, H1) \rightarrow V3(H1, H3) \rightarrow V4(H3)$ and $V2(H0, H2) \rightarrow V1(H2, H1) \rightarrow V3(H1, H3) \rightarrow V4(H3)$ respectively.

We describe the concept of an attack graph to provide background information; however, we will not use this method to generate our attack graph for this project.

Chapter 3

Attack Patterns

3.1 Threat Model and Attack Model

1. Threat Model:

- A **threat model** is a systematic approach to predicting and understanding potential threats to an organization's ecosystem. It involves identifying vulnerabilities and assessing the risks associated with them.
- In threat modeling, we consider various scenarios and potential security violations that might exploit system weaknesses. These threats can arise from accidental events, environmental factors (such as natural disasters), human negligence, or deliberate attacks.[5]
- Types of security threats include:
 - **Unstructured Threats:** These are typically executed by inexperienced individuals using easily accessible hacking tools. Despite their simplicity, they can cause significant damage to a company.
 - **Structured Threats:** These involve organized attempts to breach a specific network or organization. They come from highly motivated and technically proficient hackers.
 - **External Threats:** Originating from individuals or organizations outside the company, these threats have unauthorized access to computer systems and networks (often via the Internet or dial-up access servers).
 - **Internal Threats:** These occur due to authorized network access, whether through a server account or physical access¹

2. Attack Model:

- An **attack model** focuses on deliberate, unauthorized actions taken against a system or asset. Attacks can be classified as either **active** or **passive**:
 - **Active Attacks:** Aim to manipulate system resources or impact their operation. These attacks directly interfere with the system's functioning.

¹<https://www.geeksforgeeks.org/difference-between-threat-and-attack/>

- **Passive Attacks:** Aim to extract sensitive information from a system without affecting its resources. These attacks are more subtle and stealthy.
- Common classes of attacks include:
 - **Reconnaissance:** Gathering information about the target system.
 - **Denial of Service (DoS):** Overloading a system to disrupt its normal operation.
 - **Worms, Viruses, and Trojan Horses:** Malicious software that spreads and harms systems.
 - **Unauthorized Access:** Gaining entry to a device without proper credentials.
- Unlike threat models, attack models assume malicious intent and focus on causing damage or extracting information.
- Attacks can be initiated by outsiders (both systems and users) and are always intentional.

threat modeling helps us understand potential risks, while attack modeling focuses on deliberate actions taken by adversaries. ²Both are essential for effective security measures in any system or organization.

3.2 CAPEC

The Common Attack Pattern Enumeration and Classification (CAPEC) provides a publicly available comprehensive dictionary of known attack patterns that are commonly used by adversaries to exploit weaknesses in cyber infrastructures. The US Department of Homeland Security initially released it in 2007. The current version as of 2023 is version 3.9, which has 559 attack patterns. CAPEC attack patterns are classified into 6 “Domains” and 9 “Mechanisms” of Attack.[6]

Domains of Attack:

- Software
- Hardware
- Communication
- Supply Chain
- Social Engineering
- Physical Security

Mechanisms of Attack:

- Engage in Deceptive Interactions
- Abuse Existing Functionality
- Manipulate Data Structures
- Manipulate System Resources

²https://www.splunk.com/en_us/blog/learn/threat-modeling.html

- Inject Unexpected Items
- Employ Probabilistic Techniques
- Manipulate Timing and State
- Collect and Analyze Information
- Subvert Access Control

“Attack Patterns” are descriptions of the common attributes and approaches employed by adversaries to exploit known vulnerabilities. Attack patterns define the challenges that an adversary may face and how they go about solving them. Each attack pattern captures knowledge about how specific parts of an attack are designed and executed and gives guidance on ways to mitigate the attack’s effectiveness.

3.3 CWE

CWE(Common Weakness Enumeration)³ is a community-developed list of common software and hardware weakness types that have security ramifications. A “weakness” is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities. The CWE List and associated classification taxonomy serve as a language that can be used to identify and describe these weaknesses in terms of CWEs. In general, CWE(s) describe the root cause(s) of vulnerabilities. In the CWE hierarchy, weaknesses are organized into a tree-like structure with different levels of abstraction. Where Pillar has the highest Abstraction, the most general and Variant is the most specific.

Pillars > Classes > Base > Variant

3.3.1 Pillars

A weakness that is the most abstract type of weakness and represents a theme for all class/base/-variant weaknesses related to it. A Pillar is still technically a type of weakness that describes a mistake. One example of Pillar type Weakness is **Improper Neutralization**.⁴⁵ It refers to a security vulnerability that occurs when input data is not properly sanitized or validated before being used in a system. This vulnerability can lead to various types of attacks, such as injection attacks (e.g., SQL injection, cross-site scripting) and buffer overflows. In improper neutralization vulnerabilities, an attacker can exploit the system by injecting malicious code or unexpected data into the application, potentially gaining unauthorized access, manipulating data, or causing the system to behave unexpectedly.[10]

Here is an example of Improper Neutralization. Suppose we have a web application that constructs an SQL command using externally-influenced input from an upstream component. The application does not properly neutralize special elements that could modify the intended SQL command when it is sent to a downstream component. This vulnerability can lead to SQL injection.⁶

³<https://cwe.mitre.org/about/index.html>

⁴<https://www.martellosecurity.com/kb/mitre/cwe/79/>

⁵<https://cwe.mitre.org/data/definitions/707.html>

⁶<https://cwe.mitre.org/data/definitions/89.html>

```
SELECT * FROM users WHERE username = '$user_input';
```

If an attacker provides the input '**OR '1'='1**' the query becomes:

```
SELECT * FROM users WHERE username = '' OR '1'='1';
```

The query always evaluates to true ('1'='1'), potentially exposing sensitive user data. Here are some techniques used for neutralization: **Validation:** Checking that input/output adheres to expected formats or constraints. **Transformation:** Modifying input/output to make it safe (e.g., filtering, encoding/decoding, escaping/unescaping). **Indirect Selection:** Mapping externally-provided values to internally-controlled values. **Preventing Processing:** Avoiding direct processing of potentially malicious input. This type of weakness is generally found in various :

- Operating Systems
- Architectures
- Technologies

3.3.2 Classes

A weakness that is described in a very abstract fashion, typically independent of any specific language or technology. More specific than a Pillar Weakness, but more general than a Base Weakness. Class level weaknesses typically describe issues in terms of 1 or 2 of the following dimensions: behavior, property, and resource. **Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')**⁷ is a class Type of Weakness. The product constructs all or part of a command, data structure, or record using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify how it is parsed or interpreted when it is sent to a downstream component. It is the lack of verification of these assumptions for user-controlled input that leads to injection problems. The most important issue to note is that all injection problems share one thing in common – i.e., they allow for the injection of control plane data into the user-controlled data plane. This means that the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism. The following examples help to illustrate the nature of this weakness and describe methods or techniques .

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

This example code intends to take the name of a user and list the contents of that user's home directory. It is subject to the first variant of OS command injection. The **\$userName** variable is not checked for malicious input. An attacker could set the **\$userName** variable to an arbitrary OS command such as:

```
;rm -rf /
```

Which would result in **\$command** being:

```
ls -l /home/;rm -rf /
```

⁷<https://www.martellosecurity.com/kb/mitre/cwe/74/>

Since the semi-colon is a command separator in Unix, the OS would first execute the ls command, then the rm command, deleting the entire file system. This weakness is caused during the implementation of an architectural security tactic and It is not any language-specific.

3.3.3 Base

A weakness that is still mostly independent of a resource or technology, but with sufficient details to provide specific methods for detection and prevention. Base level weaknesses typically describe issues in terms of 2 or 3 of the following dimensions: behavior, property, technology, language, and resource. **Improper Control of Generation of Code ('Code Injection')**⁸ is one example of Base type Weakness. This weakness is typically seen in scenarios where user input or externally provided data is directly used to generate code without proper validation or sanitization. The following examples⁹ help to illustrate the nature of this weakness and describe methods or techniques.

```
$MessageFile = "messages.out";
if ($_GET["action"] == "NewMessage") {
    $name = $_GET["name"];
    $message = $_GET["message"];
    $handle = fopen($MessageFile, "a+");
    fwrite($handle, "<b>$name</b> says '$message'<hr>\n");
    fclose($handle);
    echo "Message Saved!<p>\n";
}
else if ($_GET["action"] == "ViewMessages") {
    include($MessageFile);
}
```

This example attempts to write user messages to a message file and allow users to view them. It allows users to submit arbitrary messages and names via the "NewMessage" action without proper validation, which are then directly included in a file without sanitization. This can enable attackers to inject malicious HTML or code into the message, leading to potential cross-site scripting (XSS) attacks. Additionally, the "ViewMessages" action includes the content of the message file without any validation, which can also result in XSS vulnerabilities if the file contains untrusted content. If an attacker provide a message such as:

```
name=h4x0r
message=%3C?php%20system(%22/bin/ls%20-l%22);?%3E
```

which will decode to the following:

```
<?php system("/bin/ls -l");?>
```

The programmer thought they were just including the contents of a regular data file, but PHP parsed it and executed the code. Now, this code is executed any time people view messages. These weaknesses can be found in some Interpreted Language.

⁸<https://www.martellosecurity.com/kb/mitre/cwe/94/>

⁹<https://cwe.mitre.org/data/definitions/94.html>

3.3.4 Variant

a weakness that is linked to a certain type of product, typically involving a specific language or technology. More specific than a Base weakness. Variant level weaknesses typically describe issues in terms of 3 to 5 of the following dimensions: behavior, property, technology, language, and resource. **Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')**¹⁰ is an example of variant weakness. Eval injection, also known as “code injection” or “dynamic code execution,” is a security vulnerability that occurs when untrusted data is passed to an “eval ()” or similar function in a programming language. Here is an example¹¹ of Eval Injection :

```
def main():
    sum = 0
    numbers = eval(input("Enter a space-separated list of numbers: "))
    for num in numbers:
        sum = sum + num
    print(f"Sum of {numbers} = {sum}")
main()
```

This simple script asks a user to supply a list of numbers as input and adds them together. The **eval()** function in Python evaluates a string as a Python expression and returns the result. It allows developers to dynamically execute code during runtime, providing great flexibility. However, if user-supplied input is directly passed into **eval()**, it can lead to code injection vulnerabilities. Code injection occurs when an attacker manages to insert malicious code into a program, exploiting a vulnerability in the system. In the case of Python EVAL code injection, an attacker manipulates input data to execute unintended commands, potentially compromising the security and integrity of the system. For example, a malicious user could supply the following string:

```
--import--('subprocess').getoutput('rm -r *')
```

If an attacker enters malicious input like this instead of a valid number, the **eval()** function will execute the unintended command, resulting in the deletion of files on the system. These weaknesses can be found in some Programming Languages.

- Java
- JavaScript
- Python
- Perl
- PHP
- Ruby

There exists an alternative hierarchy within the Common Weakness Enumeration (CWE) framework, organized by hardware design, with **'category'** representing the highest level of abstraction.

¹⁰<https://beaglesecurity.com/blog/vulnerability/eval-injection.html>

¹¹<https://vk9-sec.com/exploiting-python-eval-code-injection/>

3.3.5 Categories:

categories serve to group weaknesses that share a common characteristic. They provide a higher level of organization within the CWE. **Validate Inputs** is a Category Type of Weakness. Weaknesses in this category are related to the design and architecture of a system's input validation components. Frequently these deal with sanitizing, neutralizing and validating any externally provided inputs to minimize malformed data from entering the system and preventing code injection in the input data. The weaknesses in this category could lead to a degradation of the quality of data flow in a system if they are not addressed when designing or implementing a secure architecture. all three weaknesses— **Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')**, **Improper Control of Generation of Code ('Code Injection')**, and **Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')** —are categorized under "**Validate Inputs**" because they highlight the critical importance of properly validating input data to prevent security vulnerabilities and ensure the integrity, consistency, and security of software systems.[8]

3.4 CVE

CVE, short for Common Vulnerabilities and Exposures¹², is a list of publicly disclosed computer security flaws. When someone refers to a CVE, they mean a security flaw that's been assigned a CVE ID number. Security advisories issued by vendors and researchers almost always mention at least one CVE ID. CVEs help IT professionals coordinate their efforts to prioritize and address these vulnerabilities to make computer systems more secure.[7]Here are some key points about CVEs¹³:

Definition: A CVE is a dictionary of common names (CVE Identifiers) for publicly known information security vulnerabilities.

Purpose: CVEs allow users to recognize unique vulnerabilities and coordinate the development of security tools and solutions.

Assignment: CVE identifiers are assigned by a CVE Numbering Authority (CNA), which includes major IT vendors, security companies, and research organizations. Submission: Flaws can be submitted by vendors, researchers, or community members.

Details: CVE entries are brief and don't include technical data or risk information. Additional details appear in other databases. Now, Let's see how these CWE are related with CVE and CAPEC. CVE IDs are associated with specific weaknesses listed in CWE and CVEs represent specific instances of vulnerabilities that attackers can exploit using CAPEC techniques.¹⁴

Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')¹⁵ has several vulnerabilities. Like :

1. CVE-1999-0067:

- This vulnerability is a canonical example of **OS command injection**.
- A CGI program (phf) fails to properly neutralize the “—” metacharacter when invoking a phonebook program.

¹²<https://www.redhat.com/en/topics/security/what-is-cve>

¹³https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures

¹⁴https://cwe.mitre.org/documents/cwe_usage/guidance.html

¹⁵<https://cwe.mitre.org/data/definitions/74.html>

- The vulnerability is related to **improper neutralization of special elements in output** because it allows arbitrary command execution downstream¹⁶

2. CVE-2022-1509:

- This vulnerability involves **sed script syntax injection**.¹⁷
- It affects GitHub repository **hestiacp/hestiacp** prior to version 1.5.12.
- An authenticated remote attacker with low privileges can execute arbitrary code under root context.
- The vulnerability is related to **improper neutralization of special elements in output** because it affects how sed scripts are interpreted downstream¹⁸

3. CVE-2020-9054:

- This vulnerability affects multiple **ZyXEL network-attached storage (NAS)** devices.
- It allows pre-authentication **OS command injection**.
- An unauthenticated remote attacker can execute arbitrary code on a vulnerable device.
- The vulnerability is related to **improper neutralization of special elements in output** because it affects how commands are executed downstream.

4. CVE-2021-44228 (Log4Shell):

- This vulnerability affects **Apache Log4j 2**, a widely used Java logging framework.
- It allows remote code execution (RCE) due to improper handling of `${xyz}` style expressions.
- The vulnerability is related to **improper neutralization of special elements in output** because it affects how expressions are processed downstream.

The Weaknesses are also related to ATTACK PATTERNS. CWE entries describe weaknesses in software systems, such as buffer overflows, injection flaws, or improper input validation. Attack patterns provide concrete examples of how these weaknesses can be exploited by attackers. For instance, an attack pattern targeting injection may correspond to CWE-74, which represents the weakness of Improper Neutralization of Special Elements in Output Used by a Downstream Component. There are lots of CAPEC entries related to CWE- 74. Here we describe only 4 of them. Let's see how this **Common Attack Pattern Enumeration and Classification (CAPEC)** entries are related to **CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')**.

1. CAPEC-10: Buffer Overflow via Environment Variables:

- This attack pattern involves improper handling of environment variables.
- It can lead to buffer overflows, which are a form of code injection.

¹⁶<https://cwe.mitre.org/data/definitions/74.html>

¹⁷<https://huntr.com/bounties/09e69dff-f281-4e51-8312-ed7ab7606338/>

¹⁸<https://www.openCVE.io/cve/CVE-2022-36069>

- The vulnerability is related to **improper neutralization of special elements in output** because it affects how environment variables are processed downstream.
2. **CAPEC-101: Server Side Include (SSI) Injection:**
 - SSI injection occurs when user-controlled input is improperly included in server-side includes.
 - It can lead to code execution or information disclosure.
 - The vulnerability is related to **improper neutralization of special elements in output** because it affects how SSI directives are interpreted downstream.
 3. **CAPEC-105: HTTP Request Splitting:**
 - HTTP request splitting occurs when user input is not properly neutralized in HTTP headers.
 - It can lead to response splitting attacks.
 - The vulnerability is related to **improper neutralization of special elements in output** because it affects how HTTP headers are processed downstream.
 4. **CAPEC-108: Command Line Execution through SQL Injection:**
 - This attack pattern involves SQL injection leading to command execution.
 - It can occur when user input is improperly handled in SQL queries.
 - The vulnerability is related to **improper neutralization of special elements in output** because it affects how SQL queries are constructed and executed downstream.

Let's explore how these vulnerabilities are connected to The 'Base' Weakness **CWE-94: Improper Control of Generation of Code ('Code Injection')**

1. **CVE-2022-2054:**
 - **Description:** This vulnerability involves **code injection** in the GitHub repository `nuitka/nuitka` prior to version 0.9.¹⁹
 - It is related to **CWE-94** because it allows malicious strings to be executed as Python code, demonstrating improper control over code generation.
2. **CVE-2021-22204:**
 - **Description:** This vulnerability is part of a **chain** that starts with a regex issue in the EXIF processor code.
 - The regex issue does not correctly determine where a string ends (CWE-625).²⁰
 - This enables **eval injection** (CWE-95), leading to code execution.
 - It is related to **CWE-94** because it involves improper control over code generation via eval injection.

¹⁹<https://github.com/Nuitka/Nuitka/issues/1655>

²⁰https://owasp.org/www-community/vulnerabilities/Overly_Permissive_Regular_Expression

3. CVE-2020-8218:

- **Description:** This vulnerability is labeled as “code injection” in a VPN product.
 - It is related to **CWE-94** because it allows arbitrary code execution, demonstrating improper control over code generation.
4. **CVE-2008-5071, CVE-2002-1750, CVE-2008-5305, CVE-2002-1752, CVE-2002-1753, CVE-2005-1527, CVE-2005-2837, CVE-2005-1921, CVE-2005-2498, CVE-2005-3302, CVE-2007-1253, CVE-2001-1471, CVE-2002-0495, CVE-2005-1876, CVE-2005-1894, CVE-2003-0395:**
- These vulnerabilities involve various forms of **eval injection**, where user-controlled input is improperly executed as code.
 - They are all related to **CWE-94** because they demonstrate improper control over code generation, allowing arbitrary code execution.

Let’s explore how the following **Common Attack Pattern Enumeration and Classification (CAPEC)** entries are related to the **CWE-94: Improper Control of Generation of Code (‘Code Injection’)** weakness:

1. CAPEC-242: Code Injection:

- **Description:** This attack pattern involves injecting code into a target system.
- It is directly related to **CWE-94** because it demonstrates improper control over code generation, allowing arbitrary code execution.

2. CAPEC-35: Leverage Executable Code in Non-Executable Files:

- **Description:** This attack pattern leverages executable code within files that are not intended to execute code.
- It is related to **CWE-94** because it involves injecting code into unexpected locations, leading to code execution.

3. CAPEC-77: Manipulating User-Controlled Variables:

- **Description:** An adversary can override variables leveraging user-supplied, untrusted query variables directly used on the application server without any data sanitization.
- It is related to **CWE-94** because it demonstrates improper control over code generation via manipulation of user-controlled variables.

Also, **CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code (‘Eval Injection’)** involve scenarios like where an application takes input that is expected to be data, but executes it as code due to improper handling of the input. This typically occurs in situations where an application uses `eval()`-like functions that execute strings of text as code, which can lead to malicious code execution if an attacker can inject directives into these strings. These weaknesses are related to following vulnerabilities :

- **CVE-2022-2054:** The Python compiler’s use of `eval()` to execute strings as code directly corresponds to **CWE-95**, as `eval()` dynamically evaluates and executes code.

- **CVE-2021-22204:** The chain involving a regex issue in EXIF processor code leading to eval injection is an instance of CWE-95, as the improper neutralization allows for dynamically evaluated code to be manipulated.
- **CVE-2021-22205:** Similar to CVE-2021-22204, this chain involves bypassing validation to achieve eval injection, fitting the CWE-95 description.
- **CVE-2008-5071, CVE-2002-1750, CVE-2008-5305, CVE-2002-1752, CVE-2002-1753, CVE-2005-1527, CVE-2005-2837, CVE-2005-1921, CVE-2005-2498, CVE-2005-3302, CVE-2007-1253:** These all involve eval injections in various programming languages, where the `eval()` function or its equivalent is used to execute code, aligning with CWE-95.
- **CVE-2001-1471, CVE-2007-2713:** These vulnerabilities involve chains that result in eval injection, where the application's failure to properly neutralize input before it is dynamically evaluated leads to code execution.

The relationship between the CWE-95 and Attack Patterns(CAPEC-95)²¹ is that it allows the execution of the attack described in CAPEC-35. Essentially, if a system is vulnerable to CWE-95, it could be compromised by an attack leveraging the method outlined in CAPEC-35. The CAPEC entry details the prerequisites and skills required for such an attack, emphasizing the need for the ability to modify non-executable files consumed by the target software. This type of attack exploits the system's trust in its configuration and resource files, which is a vulnerability that CWE-95 could potentially expose. [12] [11] [13]

3.5 Attack Pattern

An attack pattern²² 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.

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. The consequences of an initial attack can lay the groundwork or create prerequisites for subsequent attacks.

The following details have been taken from CAPEC²³ website.

²¹<https://capec.mitre.org/data/definitions/35.html>

²²https://capec.mitre.org/documents/Attack_Patterns-Knowing_Your_Enemies_in_Order_to_Defeat_Them-Paper.pdf

²³<https://capec.mitre.org/data/definitions/1000.html>

3.5.1 Category Attack Pattern

[2] A category in CAPEC is a collection of attack patterns based on some common characteristic. More specifically, it is an aggregation of attack patterns based on effect/intent (as opposed to actions or mechanisms, such an aggregation would be a meta attack pattern). An aggregation based on effect/intent is not an actionable attack and as such is not a pattern of attack behavior. Rather, it is a grouping of patterns based on some common criteria. Here is an example

Tampering²⁴. Tampering attacks involve unauthorized modification of data or systems. This can include manipulating data or parameters in transit, altering configuration settings, or modifying files to achieve malicious objectives. **Inject Unexpected Items** attack is a type of attack that aims to control or disrupt the behavior of a target system by submitting crafted data or malicious code via an input interface.

Injecting unexpected items can have serious and wide-ranging consequences, including physical harm, technical malfunctions, data corruption, legal issues, reputation damage, and environmental impact.

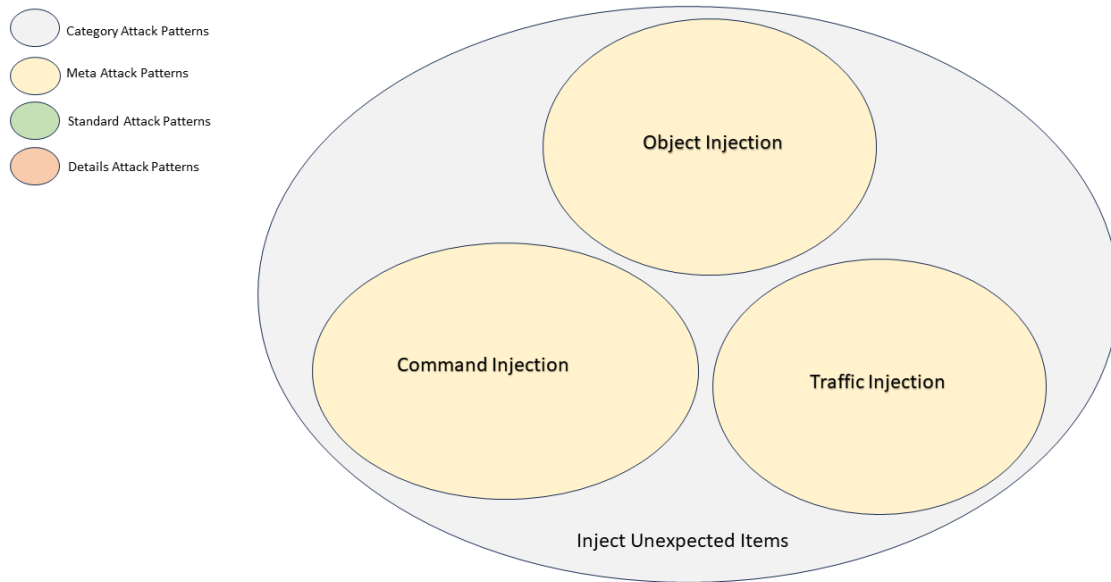


Figure 3.1: Category attack patterns

3.5.2 Meta Attack Pattern

A meta level attack pattern in CAPEC is a decidedly abstract characterization of a specific methodology or technique used in an attack. A meta attack pattern is often void of a specific technology or implementation and is meant to provide an understanding of a high level approach. A meta level

²⁴<https://csrc.nist.gov/glossary/term/tampering>

attack pattern is a generalization of related group of standard level attack patterns. Meta level attack patterns are particularly useful for architecture and design level threat modeling exercises.

A command injection²⁵ is a type of cyberattack that involves executing arbitrary commands on the host operating system through a vulnerable application. This attack differs from Code Injection, in that code injection allows the attacker to add their own code that is then executed by the application. In Command Injection, the attacker extends the default functionality of the application, which executes system commands, without the necessity of injecting code. An example of a command injection attack is when a web application allows users to upload files to the server. The application then constructs a command like this:

```
unzip $user_input
```

where **\$user_input** is the name of the file uploaded by the user. If the user uploads a valid zip file, such as **documents.zip**, the command will work as expected and unzip the file. However, if the user uploads a malicious file that contains a command separator, such as **;**, the attacker can append another command to the original one and execute it on the server. For example, if the user uploads a file named: **documents.zip; nc -e /bin/sh 192.168.1.100 4444** The command will become:

```
unzip documents.zip; nc -e /bin/sh 192.168.1.100 4444
```

This will execute two commands: the first one will unzip the file, and the second one will create a reverse shell connection to the attacker's machine, giving the attacker remote access to the server. Depending on the privileges of the application and the server configuration, the attacker can execute more harmful commands, such as installing malware, accessing sensitive data, or compromising other systems.

Another meta type attack is Traffic Injection²⁶. Traffic injection, in refers to the unauthorized or malicious insertion of traffic into a network or system. This can occur at various levels of the network stack and can have different purposes, including data interception, manipulation, or disruption. One example is DNS (Domain Name System) traffic injection, where attackers insert false DNS responses to redirect users to malicious websites, like injecting ads²⁷ into the websites that users are browsing, based on the keywords they use. The ads are generated by toolbars or browser extensions that users have installed, which have permission to display relevant ads. When users click on these ads, they are redirected to the attacker's landing page or offer.

For example²⁸, suppose a user searches for **"movie theaters"** on a search engine. The user then clicks on a link to a website that shows the movie listings and showtimes. However, the user has a toolbar installed that injects ads into the website, based on the keyword **"movie theaters"**. The ads appear as links on the original website, highlighted with the keyword. If the user clicks on one of these links, they will be redirected to another website that offers movie tickets, streaming services, or other products related to movies. The user may not realize that they have left the original website, and may be tricked into buying something or providing their personal or financial information.

²⁵<https://www.imperva.com/learn/application-security/command-injection/>

²⁶https://doc.zeropark.com/en/injection_traffic.html

²⁷<https://zeropark.com/blog/injection-traffic-for-e-commerce-advertising/>

²⁸<https://www.catchpoint.com/network-admin-guide/dns-tunneling>

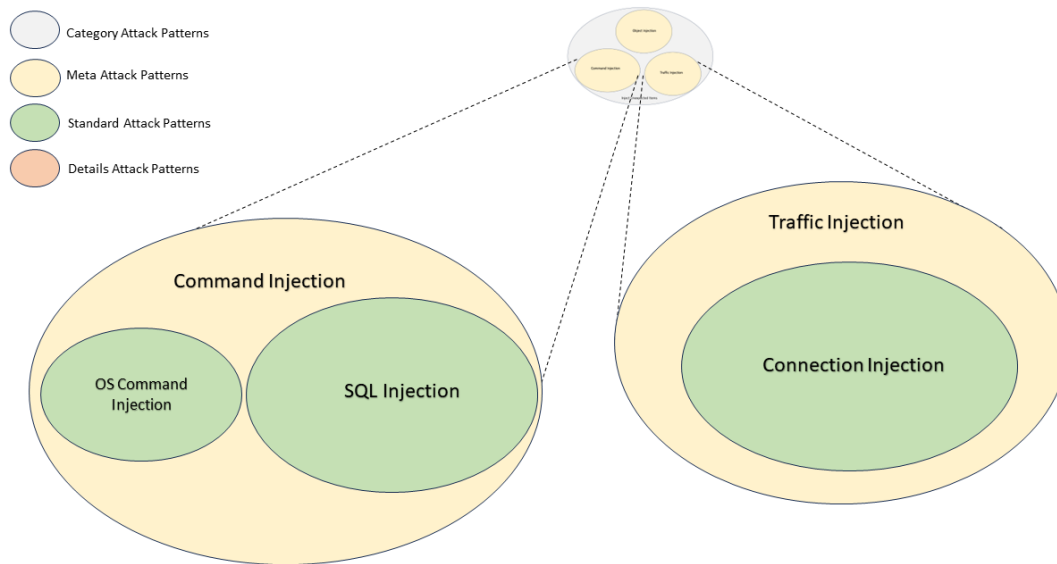


Figure 3.2: Meta Attack Pattern

3.5.3 Standard Attack Pattern

A standard level attack pattern in CAPEC is focused on a specific methodology or technique used in an attack. It is often seen as a singular piece of a fully executed attack. A standard attack pattern is meant to provide sufficient details to understand the specific technique and how it attempts to accomplish a desired goal. A standard level attack pattern is a specific type of a more abstract meta level attack pattern.

OS Command injection ²⁹ is a standard type of attack. The attacker injects malicious commands into input fields or parameters command-line interfaces, or any interface where user input is processed and executed, exploiting the system's failure to properly validate or sanitize user input. Let's take an example:

```
int main(char* argc, char** argv)
\subsection{
    char cmd[CMDMAX] = "/usr/bin/cat ";
    strcat(cmd, argv[1]);
    system(cmd);
}
```

The above program accepts a filename as a command line argument and displays the contents of the file back to the user. Let's break down the code. *int main(char* argc, char** argv)*. This main function takes command line arguments as parameters. *argc* is the number of command line arguments, and *argv* is an array of strings representing the actual arguments.

²⁹<https://owasp.org/www-community/attacks/Command-Injection>

`char cmd[CMD_MAX] = "/usr/bin/cat"` Declares a character array `cmd` with a maximum size specified by `CMD_MAX`. It is initialized with the string `"/usr/bin/cat"`.

`strcat(cmd, argv[1])` Concatenates the string at runtime which is stored in `argv[1]` (the first command line argument) to the end of the `cmd` string.

`system(cmd)` Executes the command stored in the `cmd` string using the system function. In this case, it runs the concatenated command, which is the `"/usr/bin/cat"` command followed by the first command line argument. This line effectively reads and prints the contents of the file specified as the first command line argument.

If the program runs with root privileges, the call to `system()` also executes with root privileges. If a user specifies a standard filename, the call works as expected. However, if an attacker passes a string of the form `"; rm -rf "`, then the call to `system()` fails to execute `cat` due to a lack of arguments and then plows on to recursively delete the contents of the root partition.

OS Command Injection allows attackers to execute unauthorized commands on a server, leading to unauthorized file access/modification, malware installation, and privilege escalation. It can compromise system integrity, confidentiality, and facilitate session compromise or impersonation, risking data leakage, fraudulent activities, or service disruption.³⁰

Another Standard type of attack is SQL injection.³¹ A SQL injection attack is a type of cyber attack that involves injecting malicious SQL code into a web application that uses a database. The attacker can exploit a vulnerability in the application's input validation or query construction to execute unauthorized SQL commands on the database. This can result in data theft, data manipulation, privilege escalation, or system compromise.

Here is an example of a SQL injection³² attack and how it exploits a web application. Suppose we have a vulnerable web application with a product search feature, and the application constructs SQL queries dynamically based on user input. The application's SQL query looks like this:

```
SELECT name, description
FROM products WHERE category = 'gift ';
```

Now, an attacker wants to exploit a SQL injection vulnerability in the application. They enter a malicious payload into the search field, like this: `' UNION SELECT username, password FROM users` – The application then constructs the following query:

```
SELECT name, description
FROM products WHERE category = 'gift '
UNION SELECT username, password FROM users —
```

³⁰<https://www.invicti.com/learn/os-command-injection/>

³¹<https://portswigger.net/web-security/sql-injection>

³²<https://portswigger.net/web-security/sql-injection>

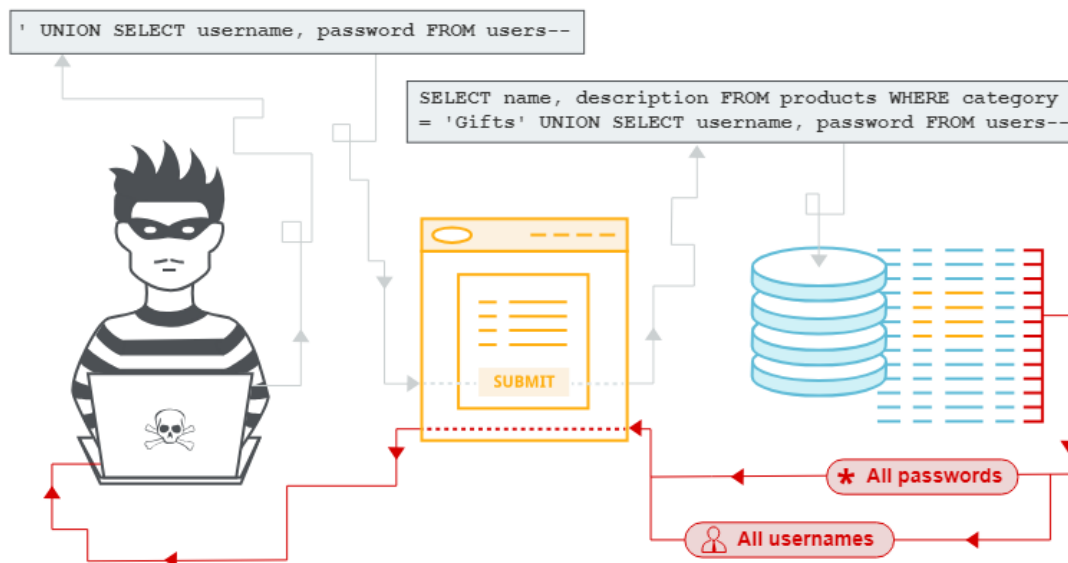


Figure 3.3: Working Principles of SQL Injection Attack

Here's how this SQL injection attack works:

Injection: The attacker enters a malicious payload (*' UNION SELECT username, password FROM users --*) into the search field.

Union: The injected payload contains a UNION clause, which merges the result set from the original query (*SELECT name, description FROM products WHERE category = 'gift'*) with the result set of another query (*SELECT username, password FROM users*).

Exfiltration: The attacker is attempting to retrieve sensitive data (in this case, usernames and passwords) from the **users** table.

Commenting: The `--` at the end of the payload is used to comment out the rest of the original query, preventing syntax errors.

SQL injections are the cause of many password and data compromises of websites and applications. Another use case for SQL injection is to hijack the intended logic of the web application. The most common example is bypassing an authentication page. Attackers may also be able to read and write files directly on the server, which can lead to backdoors being placed on the server and then the application being taken over.³³

³³<https://www.vaadata.com/blog/sql-injections-principles-impacts-exploitations-security-best-practices/>

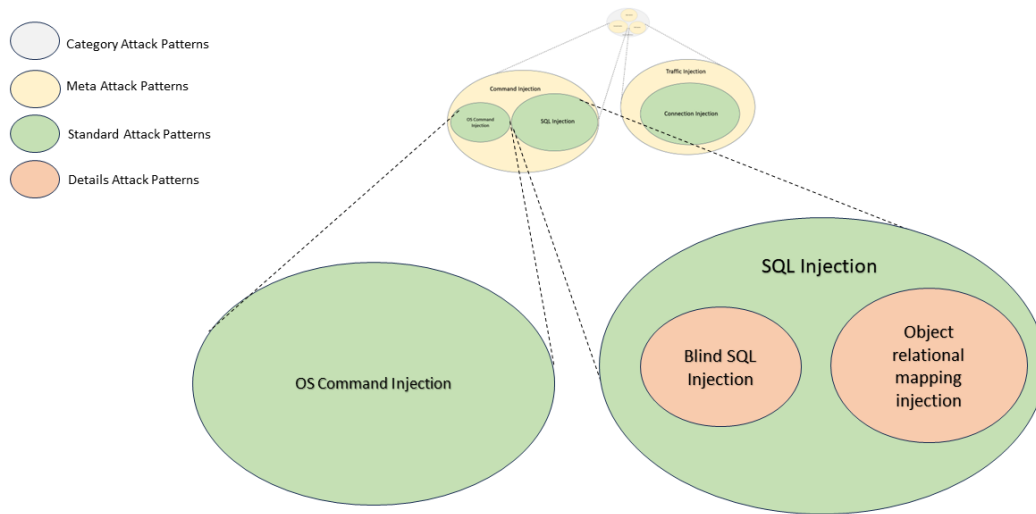


Figure 3.4: Standard Attack Pattern

3.5.4 Details Attack Pattern

A detailed level attack pattern in CAPEC provides a low level of detail, typically leveraging a specific technique and targeting a specific technology, and expresses a complete execution flow. Detailed attack patterns are more specific than meta attack patterns and standard attack patterns and often require a specific protection mechanism to mitigate actual attacks. A detailed level attack pattern often will leverage a number of different standard level attack patterns chained together to accomplish a goal. One example is **"Blind SQL injection"**³⁴ is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection. When an attacker exploits SQL injection, sometimes the web application displays error messages from the database complaining that the SQL Query's syntax is incorrect. Blind SQL injection is nearly identical to normal SQL Injection, the only difference being the way the data is retrieved from the database. When the database does not output data to the web page, an attacker is forced to steal data by asking the database a series of true or false questions. This makes exploiting the SQL Injection vulnerability more difficult, but not impossible.³⁵

Suppose we have a web application with a login page that takes a username and password and validates them against a database. The application's SQL query might look like this:³⁶

```
SELECT * FROM users
```

³⁴https://owasp.org/www-community/attacks/Blind_SQL_Injection

³⁵<https://portswigger.net/web-security/sql-injection/blind>

³⁶https://en.wikipedia.org/wiki/SQL_injection

```
WHERE username = '<username>'
AND password = '<password>';
```

Now, let's say the application is vulnerable to blind SQL injection in the username field. An attacker could exploit this vulnerability by manipulating the username field to extract information from the database indirectly. For example, suppose the attacker wants to determine if the username "admin" exists in the database. They could input a payload like this: ' ***OR 1=1 AND username = 'admin'***'— When this payload is injected into the username field, the resulting query becomes:

```
SELECT * FROM users
WHERE username = '' OR 1=1
AND username = 'admin'—'
AND password = '<password>';
```

Injection: The attacker injects a payload (*' OR 1=1 AND username = 'admin'—*) into the username field.

Condition Manipulation: The injected payload modifies the original query to include a condition (*OR 1=1*) that is always true. This effectively bypasses the password check.

Blind Extraction: The attacker is not directly retrieving data from the database. Instead, they are using conditional statements to indirectly extract information. In this case, they're trying to determine if the username "*admin*" exists in the database.

Commenting: The — at the end of the payload comments out the rest of the original query to avoid syntax errors.

If the application behaves differently (e.g., it logs the attacker in as an admin) when the injected condition is true, the attacker can deduce that the username "admin" exists in the database. Blind SQL injection can be further exploited to extract sensitive information, such as usernames, passwords, or other data, by iteratively guessing characters or values and observing the application's response.

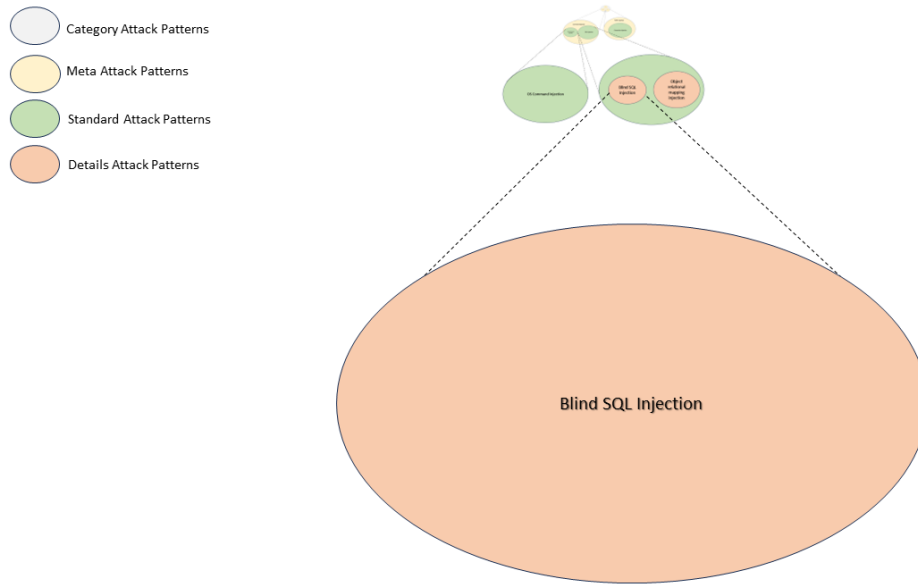


Figure 3.5: Details of the Attack Pattern of Figure 3.4

3.6 CAPEC vs. ATT&CK

CAPEC and ATT&CK³⁷³⁸ are two frameworks that describe different aspects of adversary behavior in cybersecurity. Both frameworks are based on the concept of tactics, techniques, and procedures (TTPs), which are the methods and actions that adversaries employ to achieve their objectives. CAPEC and ATT&CK are complementary and can be used together to understand the full spectrum of adversary behavior, from exploiting vulnerabilities to executing malicious activities.

3.6.1 What is CAPEC

CAPEC focuses on application security and enumerates the common exploits that adversaries use to attack vulnerable systems and it focuses more on attack patterns themselves. [9] Some characteristics of CAPEC are:

- Focus on application security
- Enumerates exploits against vulnerable systems
- Includes social engineering/supply chain
- Associated with Common Weakness Enumeration (CWE)

³⁷<https://security.stackexchange.com/questions/240478/what-is-the-difference-between-attck-and-capec>

³⁸<https://capec.mitre.org/about/attack-comparison.html>

3.6.2 What is ATT&CK

ATT&CK focuses on network defense and describes the operational phases and techniques that adversaries use to compromise and operate inside a network. ATT&CK provides a detailed view of adversarial tactics and techniques. Some characteristics of ATT&CK are:

- Focus on network defense
- Based on threat intelligence and red team research
- Provides a contextual understanding of malicious behavior
- Supports testing and analysis of defense options

Chapter 4

Proposed Methodology

4.1 Methodology

In this section, we outline the methodology employed for designing and implementing our system for analyzing attack graphs in network security. The system is divided into four main components:

1. Conversion of Input Data:

- We begin by obtaining input data from a CSV file containing network information. This file includes details about network nodes, connections, services, and vulnerabilities.
- The input data serves as the foundation for subsequent steps in our analysis.

2. Creation of the Network Graph:

- Using the Python library NetworkX, we analyze the input data and construct the network graph.
- Each node in the graph represents a network component (e.g., host, service), and edges represent connections between them.
- We consider both direct and indirect connections to capture the overall network topology.

3. Finding Possible Direct Paths:

- After creating the network graph, we focus on identifying direct service access paths between any two hosts that are directly connected.
- If a direct path exists, we extract and return the path information.
- This step helps us understand immediate communication channels within the network.

4. CVE and Patterns Mapping:

- To enhance our analysis, we utilize an additional CSV file containing mappings between tools, vulnerabilities, and attack patterns.
- Given specific vulnerabilities associated with particular services, we map them to relevant attack patterns.

- This mapping aids in identifying common attack strategies based on known vulnerabilities.

By following this methodology, we aim to create an efficient and effective system for analyzing attack graphs, understanding network vulnerabilities, and improving security measures.[3][4]

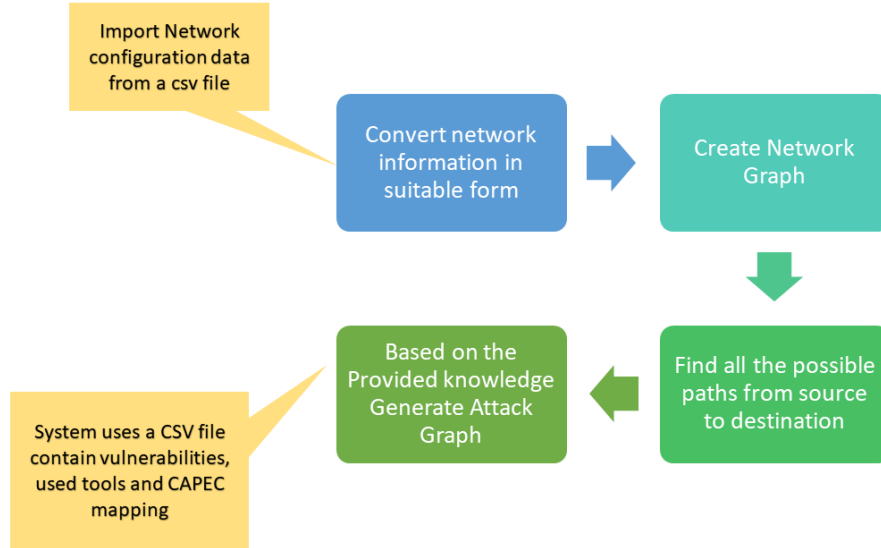


Figure 4.1: Overview of the Proposed Methodology

4.2 Algorithm

The core of our research is an algorithm designed to identify potential attack paths in a network based on the existing vulnerabilities and attack patterns. The algorithm operates in several stages:

Data Input: The algorithm begins by reading the network structure and vulnerability data from CSV files using the pandas library in Python. This data includes the hosts, services running between them, known vulnerabilities associated with these services, and tools used by the hosts.

Graph Construction: Using the NetworkX library, the algorithm constructs a directed graph representing the network. Each node in the graph represents a host, and each edge represents a service running between two hosts. The edge attributes include the service's known vulnerabilities and the tools used by the hosts.

Attack Path Identification: The algorithm performs a depth-first search (DFS) from the source host to the destination host. For each edge in the DFS path, the algorithm calls the `return_attack_path()` function. If this function returns True, indicating a potential attack path, the edge is added to a new graph representing the attack paths in the network.

Graph Visualization: Finally, the algorithm uses NetworkX and Matplotlib to visualize the attack paths. Each node in the graph represents a host, each edge represents a potential attack

path, and the edge labels represent the services and their associated vulnerabilities.

In the following sections, we will delve into a detailed discussion of our algorithm and its application to all the functions involved in our research.

This following algorithm assumes that the CSV file can be read row by row, and that each row can be indexed by column names. The specific implementation details may vary depending on the programming language and libraries used. Also, the eval function used in the original Python code is language-specific and may not have a direct equivalent in all languages.

Algorithm 1 Read_From_CSV

- 1: **Input:** csv_filename (a string representing the name of the CSV file)
 - 2: **Output:** new_source, new_destination, new_Host, new_Service, new_Vulnerability, new_Tools
 - 3:
 - 4: Initialize an empty dictionary for each of the following: new_Host, new_Service, new_Vulnerability, new_Tools
 - 5: Initialize empty strings for new_source and new_destination
 - 6: Open the CSV file with name csv_filename
 - 7: For each row in the CSV file, do the following:
 - 8: 1. Get the value of 'Category' column and assign it to variable 'category'
 - 9: 2. Get the value of 'Key' column and assign it to variable 'key'
 - 10: 3. If 'category' is in ['Host', 'Service', 'Vulnerability', 'Tools'], evaluate the value of 'Value' column and assign it to variable 'value'. Otherwise, directly assign the value of 'Value' column to 'value'
 - 11: 4. If 'category' is 'Host', add an entry to new_Host with 'key' as the key and 'value' as the value
 - 12: 5. If 'category' is 'Service', add an entry to new_Service with 'key' as the key and 'value' as the value
 - 13: 6. If 'category' is 'source', assign 'value' to new_source
 - 14: 7. If 'category' is 'destination', assign 'value' to new_destination
 - 15: 8. If 'category' is 'Vulnerability', add an entry to new_Vulnerability with 'key' as the key and 'value' as the value
 - 16: 9. If 'category' is 'Tools', add an entry to new_Tools with 'key' as the key and 'value' as the value
 - 17: Return new_source, new_destination, new_Host, new_Service, new_Vulnerability, new_Tools
-

The following algorithm aims to determine the tools associated with a specific host. Given a dictionary of tool instances, each containing information about hosts and tool names, the algorithm iterates through the instances. If the specified host matches the first element of an instance, the corresponding tool name is added to the list of tools used by that host. The output is the list of tools associated with the given host.

Algorithm 2 Get_Tools_Used

- 1: **Input:** host (a string representing the host), Tools (a dictionary where each key is a tool and the value is a list of instances, each instance is a tuple where the first element is a host and the second element is a tool name)
 - 2: **Output:** tools_used (a list of tools used by the given host)
 - 3: Initialize an empty list for tools_used
 - 4: **For each** tool **and its instances in** Tools **do the following:**
 - 5: **For each** instance **in instances,** **do the following:**
 - 6: **If** the first element of the instance is equal to the host, **add** the second element of the instance to tools_used
 - 7: Return tools_used
-

The following algorithm only considers “direct_host_service” . The original function also includes logic to check for service access via another host, which is represented by the “via_host_services” list. However, this part is not included in the algorithm above because “via_host_service” is not currently being used for research as it would be more complicated.

Algorithm 3 Find_Accessible_Services

- 1: **Input:** src_host (a string representing the source host), dst_host (a string representing the destination host), Host (a dictionary where each key is a host and the value is a list of services), Service (a dictionary where each key is a service and the value is a list of instances, each instance is a tuple of two hosts), Vulnerability (a dictionary where each key is a service and the value is a list of vulnerabilities), Tools (a dictionary where each key is a tool and the value is a list of instances, each instance is a tuple where the first element is a host and the second element is a tool)
 - 2: **Output:** direct_access_services (a list of tuples, each tuple contains a service, its vulnerabilities, and the tools used by the source host), via_host_services (a list of tuples, each tuple contains a service, its vulnerabilities, and the tools used by the source host)
 - 3: Initialize an empty list for direct_access_services and via_host_services
 - 4: **For each** service **and its instances in** Service **do the following:**
 - 5: **For each** instance **in instances,** **do the following:**
 - 6: **If** the instance is equal to the tuple (src_host, dst_host), **do the following:**
 - 7: **Get** the vulnerabilities of the service from Vulnerability, if any, and assign it to variable ‘vulnerabilities’
 - 8: **Get** the tools used by the src_host from Tools using the Get_Tools_Used algorithm and assign it to variable ‘tools_used’
 - 9: **Append** the tuple (service, vulnerabilities, tools_used) to direct_access_services
 - 10: Return direct_access_services
-

note that this algorithm assumes that the ‘database.csv’ file can be read line by line, and that each line can be split into a list of values.

Algorithm 4 Find_Patterns

```
1: Input:direct_access_services (a list of tuples, each tuple contains a service, its vulnerabilities,
   and the tools used by the source host)
2: Output: attack_patterns (a list of attack patterns)
3: Initialize an empty list for attack_patterns
4: Open the file 'database.csv'
5: Read the first line of the file (header) and ignore it
6: For each remaining line in the file, do the following:
7:   Split the line into a list of values and assign it to variable 'row'
8:   For each service in direct_access_services, do the following:
9:     Get the vulnerabilities and tools used from the service and assign them to variables 'vul-
       nerabilities' and 'tools_used', respectively
10:    For each vulnerability in vulnerabilities, do the following:
11:      If the vulnerability is in row[1] and row[0] is in tools_used, append row[3] to at-
        tack_patterns
12:      If the vulnerability is in row[1] and row[0] is "NA", append row[3] to attack_patterns
13: Return attack_patterns
```

The following algorithm aims to find an attack path between a source host and a destination host. It relies on direct access services and detected attack patterns. If an attack path exists, it constructs a directed graph with edge labels representing the attack patterns. The output indicates whether an attack path is found and provides relevant edge labels.

Algorithm 5 Return_Attack_Path

- 1: **Input:** src_host (a string representing the source host), dst_host (a string representing the destination host), Host (a dictionary where each key is a host and the value is a list of services), Service (a dictionary where each key is a service and the value is a list of instances, each instance is a tuple of two hosts), Vulnerability (a dictionary where each key is a service and the value is a list of vulnerabilities), Tools (a dictionary where each key is a tool and the value is a list of instances, each instance is a tuple where the first element is a host and the second element is a tool)
 - 2: **Output:** A boolean indicating whether attack patterns are detected, and if so, the edge labels of the attack path
 - 3: Call the Find_Accessible_Services algorithm with src_host, dst_host, Host, Service, Vulnerability, Tools as inputs, and assign the first returned value to variable 'direct_access_services'
 - 4: Call the Find_Patterns algorithm with direct_access_services as input, and assign the returned value to variable 'attack_patterns'
 - 5: **If** attack_patterns is not empty, **do the following:**
 - 6: Initialize a directed graph 'G'
 - 7: Add src_host and dst_host as nodes to 'G'
 - 8: Add an edge from src_host to dst_host to 'G', with 'attack_patterns' as the edge attribute
 - 9: Get the edge attributes of 'G' with 'attack_patterns' as the key, and assign it to variable 'edge_labels'
 - 10: **Return** True, edge_labels
 - 11: **If** attack_patterns is empty, **return** False, None
-

The “DFS All Paths” algorithm aims to find all paths from a source node to a destination node in a given graph. It uses depth-first search (DFS) and maintains a set of visited nodes and a path list. If a path is found, it yields it as output.

Algorithm 6 DFS_All_Paths

- 1: **Input:** graph (a graph data structure), source (a node in the graph), destination (a node in the graph), visited (a set of nodes that have been visited, optional), path (a list of nodes representing the current path, optional)
 - 2: **Output:** All paths from source to destination in the graph
 - 3: **If** visited is not provided, initialize visited as an empty set
 - 4: **If** path is not provided, initialize path as a list containing only the source
 - 5: Add source to visited
 - 6: **If** source is equal to destination, **yield** path
 - 7: **for** each neighbor of source in the graph **do**
 - 8: **If** neighbor is not in visited, **do the following:**
 - 9: Recursively call DFS_All_Paths with graph, neighbor, destination, a copy of visited, and path appended with neighbor
 - 10: **end for**
-

The following algorithm constructs a visualization of the attack path from a source host to a

destination host. It creates a directed graph based on services and instances, identifies valid attack paths, and displays them with edge labels representing the associated services. The output provides a visual representation of the path and relevant service information.

Algorithm 7 Return_Attack_Path_Visualization

```

1: Input: source (a string representing the source host), destination (a string representing the
   destination host), Host (a dictionary where each key is a host and the value is a list of services),
   Service (a dictionary where each key is a service and the value is a list of instances), Vulnerability
   (a dictionary where each key is a service and the value is a list of vulnerabilities), Tools (a
   dictionary where each key is a tool and the value is a list of instances)
2: Output: A visualization of the attack path from source to destination
3: Initialize a directed graph  $G$ 
4: Add each host in Host as a node to  $G$ 
5: for each service and its instances in Service do
6:   for each instance in instances do
7:     Add an edge from the first host in the instance to the second host in the instance to  $G$ ,
       with 'service' as the edge attribute
8:   end for
9: end for
10: Initialize a new directed graph  $new\_graph$ 
11: for each path from source to destination in  $G$  found by the DFS_All_Paths algorithm do
12:   for each index  $i$  from 0 to the length of path - 2 do
13:     Call the Return_Attack_Path algorithm with  $path[i]$ ,  $path[i + 1]$ , Host, Service, Vulner-
       ability, Tools as inputs, and assign the returned value to variable  $attack\_path\_result$ 
14:     if the first element of  $attack\_path\_result$  is True then
15:       Add an edge from  $path[i]$  to  $path[i + 1]$  to  $new\_graph$ , with 'service' as the edge
       attribute and the value as the string representation of the second element of  $attack\_path\_result$ 
       concatenated with the 'service' attribute of the edge from  $path[i]$  to  $path[i + 1]$  in  $G$ 
16:     end if
17:   end for
18: end for
19: Draw  $new\_graph$  with increased spacing between nodes
20: Get the edge attributes of  $new\_graph$  with 'service' as the key, and assign it to variable
    $edge\_labels$ 
21: Draw the edge labels of  $new\_graph$ 
22: Show the title 'Path from source to destination'
23: Show  $new\_graph$ 

```

Chapter 5

Experiments

5.1 Objective

The primary aim of this research is to construct an intricate and adaptable attack graph based on established vulnerabilities and attack patterns. This endeavor entails the meticulous identification and analysis of plausible attack routes within a network, taking into account the intricate interplay among various vulnerabilities and the sequential exploitation methods employed by malicious actors. Through the visualization of these attack routes, our goal is to furnish a lucid comprehension of the security stance of the system, facilitating proactive measures to mitigate potential threats. This approach not only augments the efficacy of vulnerability management but also contributes significantly to the evolution of more resilient intrusion detection and prevention systems. Ultimately, this research endeavors to elevate the overall security standards of networks and systems, thereby diminishing the exposure to cyber threats.

5.2 Hypothesis

We posit that the existence of vulnerabilities within services hosted on network nodes significantly increases the likelihood of successful cyber attacks. We propose that services with known vulnerabilities are more susceptible to exploitation, leading to potential attack paths within the network. Furthermore, we conjecture that the use of specific tools by network nodes can influence the formation of these attack paths. For instance, nodes employing certain tools might have the ability to intercept and analyze network traffic, thereby affecting the trajectory of attack paths. The choice of source and destination nodes in the network plays a crucial role in shaping these paths; we hypothesize that attack paths will be influenced by the services running on these nodes and the associated vulnerabilities. These hypotheses will be further explored and tested in our study.

The dataset utilized in this study originates from two primary sources. The initial source comprises a CSV file delineating the configuration of a compact network, encompassing host details, inter-host services, associated vulnerabilities, and the tools utilized by each host. This network structure is represented in the following manner:

Hosts: Presented as a dictionary with host identifiers as keys and lists of services running on each host as values. **Services:** Represented as a dictionary with service identifiers as keys and

lists of tuples indicating the hosts between which the respective service operates. **Vulnerabilities:** Structured as a dictionary with service identifiers as keys and lists of known vulnerabilities (CVE IDs) linked to each service. **Tools:** Organized as a dictionary with tool identifiers as keys and lists of tuples denoting the hosts employing the respective tool along with its function.

The second dataset, another CSV file, supports our proposed model by establishing a mapping between vulnerability CVE IDs, associated attack patterns, and the tools employed to exploit these vulnerabilities. This mapping is fundamental for comprehending potential attack paths within the network based on existing vulnerabilities and attack patterns.

The amalgamation of these datasets offers a holistic perspective on the network’s security posture, facilitating the identification and analysis of potential attack paths. These datasets undergo rigorous processing and analysis employing various network security analysis methods and techniques, details of which are expounded upon in subsequent sections of this project.

5.3 Experimental Setup

The experimental setup for this research involves a two-stage process. In the first stage, we utilized the Neo4j database to store the details of the graph, which includes elements such as the service, service instance, host, service access instance, vulnerability, and attacker node. Neo4j, being a graph database, provides an efficient platform for storing and querying graph data, making it an ideal choice for this stage of the experiment.

In the second stage, we used the NetworkX library in Python to visualize the graph data. NetworkX is a powerful tool for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It allowed us to effectively visualize the network structure and understand the interconnections between different nodes.

To identify potential attack paths based on the input network configuration, we developed an algorithm. This algorithm takes into account the network structure, the vulnerabilities associated with each service, and the tools used by the hosts. The identification of these attack paths is crucial for understanding the security posture of the network and for developing effective strategies for vulnerability management and intrusion detection.

Additionally, we used the pandas library in Python to read data from CSV files. Pandas provide data structures and data analysis tools that are well-suited for manipulating numerical tables and time series data. This enabled us to efficiently process the input data and transform it into a format suitable for our experiment.

The combination of these tools and techniques provided a robust experimental setup for our research. It allowed us to effectively analyze the network’s security posture, identify potential attack paths, and visualize the results in a clear and understandable manner.

5.4 Example

Now we will take an Example and dig into deeper how our system works on a given network input from CSV file.

In this example, we have a network with four hosts (h1, h2, h3, h4), three services (s1, s2, s3), and two tools (t1, t2). The services are distributed among the hosts as follows:

h1 does not provide any services. h2 provides service s1. h3 provides service s2. h4 provides services s3, s1, and s2. The services between hosts are as follows:

Service s1 is provided from h1 to h2 and from h2 to h4. Service s2 is provided from h4 to h3. Service s3 is provided from h1 to h4. Each service has associated vulnerabilities, and each host uses certain tools. Our goal is to find an attack path from h1 (source) to h3 (destination).

What Our System Does

1.Create a Directed Graph: The system first creates a directed graph based on the given input. Each node in the graph represents a host, and each directed edge represents a service from one host to another. For example, the edge for service s1 goes from h1 to h2 the edge for service s3 goes from h1 to h4 the edge for service s1 goes from h2 to h4 the edge for service s2 goes from h4 to h3

2.Perform DFS from Source to Destination: The system then performs a Depth-First Search (DFS) from the source host (h1) to the destination host (h3). DFS is a graph traversal method that explores as far as possible along each branch before backtracking. So, it first goes from h1 to h2 via s1, then from h2 to h4 via s1, and finally from h4 to h3 via s2. There are two paths to reach h3.

Path 1 $h1 \rightarrow s1 \rightarrow h2 \rightarrow h4 \rightarrow s2 \rightarrow h3$ **Path 2** $h1 \rightarrow s3 \rightarrow h4 \rightarrow s2 \rightarrow h3$

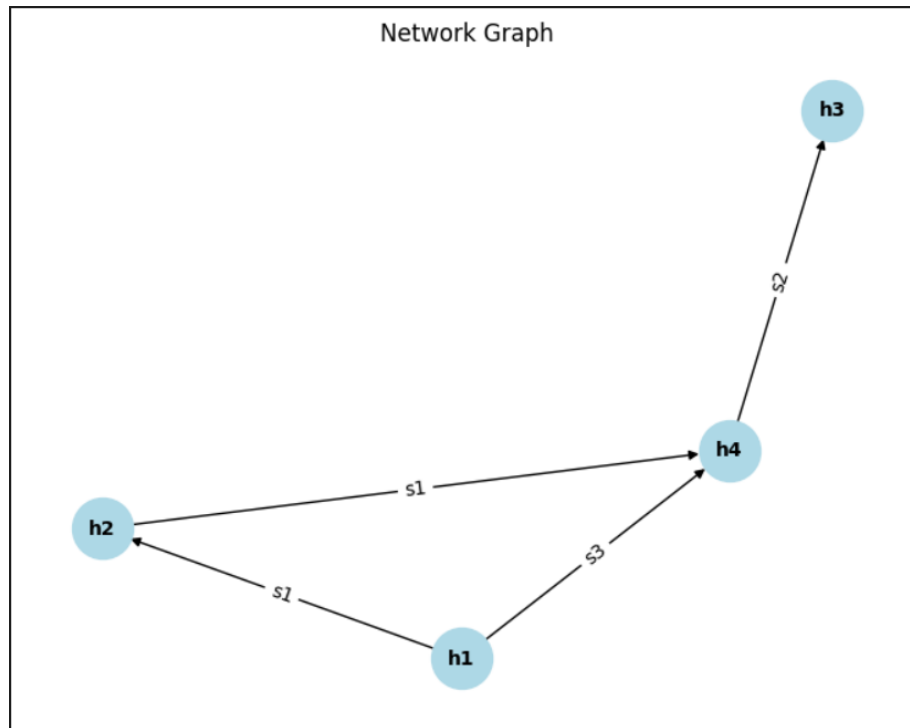


Figure 5.1: System Model

3.Detection of Attack Patterns:

If the service running between two connected hosts in a path has vulnerabilities or if the hosts use certain tools, we then check for attack patterns. This information is obtained from another CSV file mapping attack patterns with vulnerabilities and tools. If a vulnerability is detected or specific tools are used, it indicates a potential attack pattern. For example: The algorithm then identifies potential attack patterns by checking the vulnerabilities associated with each service and the tools

used by the source host. Based on the provided input data here's how our algorithm would process each path:

Read Vulnerability and Tools Data: Our system reads the vulnerability and tools mapping from the CSV file, which contains information about which tools can exploit specific vulnerabilities.

Identify Direct Access Services: The algorithm identifies services that can be directly accessed from the source host to the destination host.

Check Vulnerabilities and Tools: For each service, the algorithm checks the associated vulnerabilities and tools used. For example, Service s3 has the vulnerability CVE-2022-21668, and the tool t2 (high_request_generator) can exploit it.

Mapping with Attack Patterns: The system Maps the identified vulnerabilities and tools with the attack patterns stored in the CSV file. The algorithm would then visualize the attack paths using a directed graph, with nodes representing hosts and edges representing services. Each edge would be labeled with the service, vulnerabilities, and detected attack patterns based on the CSV mapping.

Detected Attack Patterns: For Service s3: Flooding; Matches the vulnerability CVE-2022-21668 and the tool t2 (high_request_generator). Excessive Allocation: Also matches the vulnerability CVE-2022-21668 between host h1 and h4 .

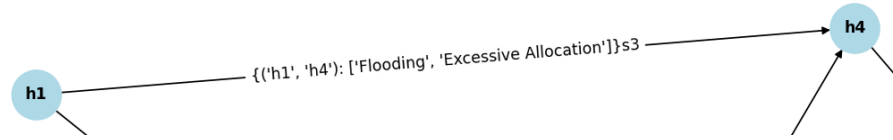


Figure 5.2: Detected Attack Patterns Between H1 to H4

For Service s1: Identity Spoofing; Matches the vulnerability CVE-2022-35248 between h1 to h2 and h2 to h4.

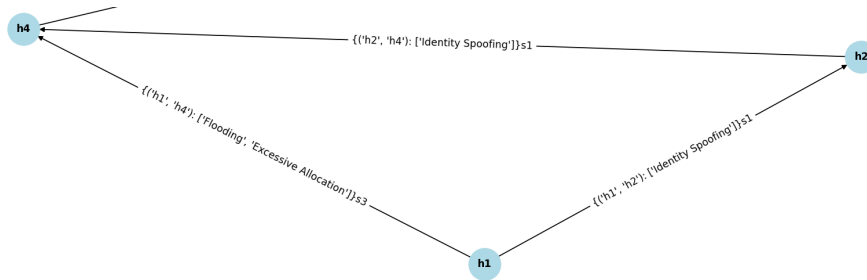


Figure 5.3: Detected Attack Patterns Between H1 to H2 and H2 to H4

For Service s2: Protocol Analysis: Matches the vulnerability CVE-2001-1546 and the tool t1 (sniffer) between h4 to h3.

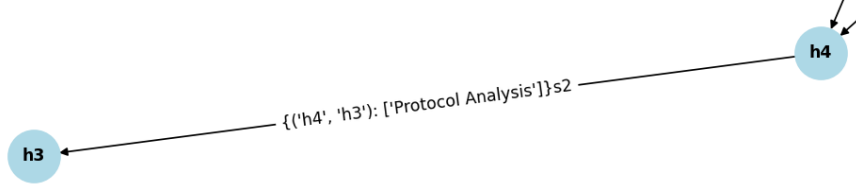


Figure 5.4: Detected attack patterns Between H4 to H3

Visualization: Finally, if attack patterns are detected, the algorithm generates a visualization of the attack path. This visualization highlights the potential vulnerabilities exploited along the path from the source host (h1) to the destination host (h3), considering the available services, vulnerabilities, and tools in the network. So for this network, Our system found Two attack paths which are shown in below :

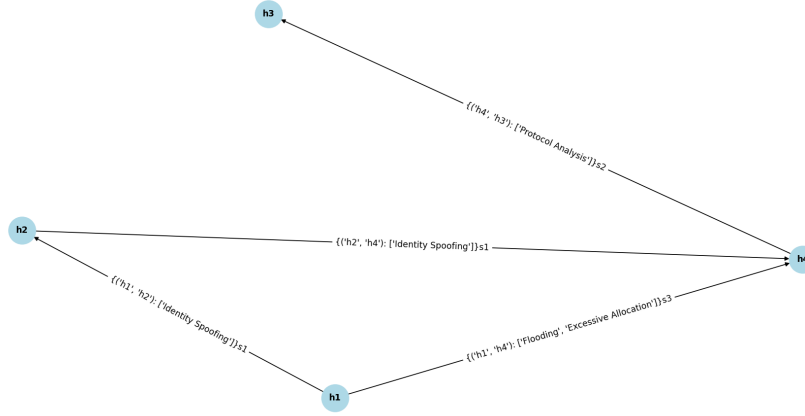


Figure 5.5: System Model

5.5 Limitations

While our research has provided valuable insights and advanced our understanding of the problem, there are several limitations that should be acknowledged:

Scalability: Our algorithms may struggle to handle large-scale networks with a high number of hosts, services, and connections efficiently. **Complexity:** The algorithms' complexity can increase significantly when dealing with various vulnerabilities, tools, and attack patterns, making it challenging to manage and optimize for performance.

Dependency on Data Quality: The accuracy and reliability of our algorithms heavily depend on the quality of the input data, including the completeness and correctness of information about hosts, services, vulnerabilities, and tools. Inaccurate or incomplete data can lead to misleading results and interpretations.

Assumptions: Our algorithms make certain assumptions about the network structure, behavior of hosts and services, and the nature of vulnerabilities and attacks. These assumptions may not always hold true in real-world scenarios, potentially resulting in inaccuracies in the analysis.

Limited Scope of Vulnerabilities and Tools: Our algorithms focus on a specific set of vulnerabilities and tools, potentially overlooking emerging threats or variations in attack techniques. This limitation can impact their ability to detect and mitigate new and evolving security risks effectively.

Performance Overhead: Some of our algorithms introduce performance overhead, particularly in terms of computational resources and execution time. This overhead can become a limitation in environments where real-time or near-real-time analysis is required.

Single Point of Failure: Certain aspects of our algorithms, such as the dependency on a central database or repository for vulnerability and tool information, create a single point of failure. Any issues or downtime with this central system can disrupt the entire analysis process.

Lack of Adaptability: The lack of adaptability in our algorithms extends to their inability to effectively handle dynamic network environments, characterized by frequent changes in hosts, services, vulnerabilities, and tools. This lack of adaptability hinders their efficacy in meeting evolving network security demands. Moreover, the dynamic nature of these changes necessitates corresponding updates to the input files, ensuring they reflect the current state of the network, including the number of hosts, vulnerabilities, services, tools, and other relevant parameters. This requirement for manual adjustment can introduce delays and potential errors, further highlighting the challenge posed by the algorithms' limited adaptability to dynamic contexts.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this research, we embarked on a journey into the intricate world of attack graphs and their associated attack paths. Our exploration began with a comprehensive definition, supported by illustrative examples. We delved into the specifics of Common Attack Pattern Enumeration and Classification (CAPEC) attack patterns, meticulously detailing their various types through diagrams and practical scenarios. Additionally, we established a clear understanding of Common Weakness Enumeration (CWE) and its typology, emphasizing the interplay between CAPEC, CWE, and Common Vulnerabilities and Exposures (CVE).

Having laid this foundational groundwork, we proceeded to develop an approach for generating attack paths from a given network schema. Our system ingeniously leveraged a CSV file as the initial schema, which served as the blueprint for constructing a comprehensive network graph. Employing a Depth-First Search (DFS) algorithm, we meticulously identified all possible attack paths within the network.

The output materialized in the form of a graph, each path meticulously labeled with the corresponding attack pattern. This visual representation proved invaluable in pinpointing potential entry points for attackers within the system. Our project not only presented algorithms for this novel framework but also subjected them to rigorous analysis. Through empirical evaluations, we demonstrated how our approach provides a more nuanced understanding of a network's security posture, thereby enabling more effective monitoring and mitigation strategies.

However, our research also acknowledges several limitations, including scalability challenges, dependency on data quality, and performance overhead. These factors can impact the efficacy and efficiency of our algorithms, especially in large-scale networks with dynamic environments. Future work should focus on addressing these limitations through the development of multi-threaded analysis capabilities and the creation of more precise and scalable mappings of vulnerabilities and attack patterns.

6.2 Future Work

While our findings and methods have contributed to identifying attack paths based on patterns, several challenges remain. In this section, we outline potential avenues for future research and system enhancements

Multi-Threaded Analysis

Our current system focuses on analyzing attack paths for a single service at a time. However, in real-world scenarios, hosts often use multiple services concurrently. To address this limitation, we propose designing a multi-threaded system capable of analyzing multiple services simultaneously. This would improve efficiency and accuracy.

Complex Mapping of Vulnerabilities and Patterns

The existing mapping between tools, vulnerabilities, and attack patterns relies on a handcrafted CSV dataset. While useful, it lacks precision and scalability. Future work involves creating a more comprehensive and dynamic mapping. We can scrape vulnerability and pattern data from authoritative sources such as the CAPEC (Common Attack Pattern Enumeration and Classification) and CWE (Common Weakness Enumeration) websites. Storing this complex mapping in a multidimensional array or graph structure would allow for more accurate and adaptable analysis.

Dynamic Node Changes during Attack Path Discovery

Our assumption that there is a fixed source (src) and destination (dst) at the beginning of the attack path finding process may not hold in dynamic environments. To address this, we need to handle changes in network conditions during the attack path discovery phase. Nodes may be added, removed, or reconfigured dynamically, affecting the attack paths.

Expanding Attack Paths with Additional Vulnerable Information

After identifying attack patterns between two hosts, the subsequent exploitation should consider more information about the newly exploited host. We propose incorporating additional vulnerable information into the attack graph dynamically. This information becomes a prerequisite condition for exploiting the next host in the path.

Neo4j Database and Cypher Queries

For efficient storage and querying of the attack graph and mapping, consider using a graph database such as Neo4j. Cypher queries, specifically designed for graph databases, can enhance the performance of attack path discovery compared to Python-based graph queries.

Bibliography

- [1] M. Barik, A. Sengupta, and C. Mazumdar. Attack graph generation and analysis techniques. *Defence Science Journal*, 66(6):559–567, Oct. 2016.
- [2] E. Fernandez, J. Pelaez, and M. Larrondo-Petrie. Attack patterns: A new forensic and design tool. In *Advances in Digital Forensics III: IFIP International Conference on Digital Forensics, National Centre for Forensic Science, Orlando, Florida, January 28-January 31, 2007 3*, pages 345–357. Springer, 2007.
- [3] C. Hankin, P. Malacaria, et al. Attack dynamics: An automatic attack graph generation framework based on system topology, capec, cwe, and cve databases. *Computers & Security*, 123:102938, 2022.
- [4] K. Kanakogi, H. Washizaki, Y. Fukazawa, S. Ogata, T. Okubo, T. Kato, H. Kanuka, A. Hazeyama, and N. Yoshioka. Tracing capec attack patterns from cve vulnerability information using natural language processing technique. 2021.
- [5] S. Lagraa, M. Husák, H. Seba, S. Vuppala, R. State, and M. Ouedraogo. A review on graph-based approaches for network security monitoring and botnet detection. *International Journal of Information Security*, 23:119–140, 2024.
- [6] T. Li, E. Paja, J. Mylopoulos, J. Horkoff, and K. Beckers. Security attack analysis using attack patterns. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–13. IEEE, 2016.
- [7] D. E. Mann and S. M. Christey. Towards a common enumeration of vulnerabilities. In *2nd Workshop on Research with Security Vulnerability Databases, Purdue University, West Lafayette, Indiana*, page 9, 1999.
- [8] B. Martin. Common vulnerabilities enumeration (cve), common weakness enumeration (cwe), and common quality enumeration (cqe) attempting to systematically catalog the safety and security challenges for modern, networked, software-intensive systems. *ACM SIGAda Ada Letters*, 38(2):9–42, 2019.
- [9] B. Martin. Common vulnerabilities enumeration (cve), common weakness enumeration (cwe), and common quality enumeration (cqe) attempting to systematically catalog the safety and security challenges for modern, networked, software-intensive systems. *ACM SIGAda Ada Letters*, 38(2):9–42, 2019.

- [10] R. A. Martin and S. Barnum. Common weakness enumeration (cwe) status update. *ACM SIGAda Ada Letters*, 28(1):88–91, 2008.
- [11] R. Miyata, H. Washizaki, K. Sumoto, N. Yoshioka, Y. Fukazawa, and T. Okubo. Identifying missing relationships of capec attack patterns by transformer models and graph structure. In *2023 IEEE/ACM 1st International Workshop on Software Vulnerability (SVM)*, pages 14–17, 2023.
- [12] F. O. Sonmez, C. Hankin, and P. Malacaria. Attack dynamics: An automatic attack graph generation framework based on system topology, capec, cwe, and cve databases. *Computers & Security*, 123:102938, 2022.
- [13] T. Tsuchida, R. Miyata, H. Washizaki, N. Yoshioka, and Y. Fukazawa. Automatic detection of abstract–concrete relationships between attack patterns of att&ck and capec with fine-tuned bert. In *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*, pages 589–590, 2023.