TaintMonkey: Dynamic Taint Analysis of Python Web Applications Using Monkey Patching

Carter Chew
carterkylechew@gmail.com

Aarav Parikh
aaravp1223@gmail.com

Shayan Chatiwala
shayan.chatiwala@gmail.com

Aiden Chen
aidenchen.contact@gmail.com

Sebastian Mercado
simercado07@gmail.com

Benson Liu*
bensonhliu@gmail.com

New Jersey's Governor's School of Engineering and Technology
July 24, 2025

*Corresponding Author

***Abstract*—Web application attacks have become more common in recent years, creating demand for robust, scalable vulnerability detection tools. Many current tools rely on static analysis, which has theoretical limitations, meaning it can only provide approximations of program behavior. Although dynamic analysis is oftentimes more practical than static analysis, there is a lack of dynamic taint analysis tools for interpreted languages. To address this issue, this paper presents TaintMonkey, a dynamic taint analysis library for Python Flask applications. To perform taint analysis, TaintMonkey uses monkey patching, a feature of some interpreted languages that allows functionality to be added to programs without editing source code. By writing plugins that monkey patch critical functions in their web applications, developers can easily add taint analysis instrumentation. TaintMonkey also includes a fuzzer, which generates randomized payloads to test web applications' security against different kinds of attacks. To assess TaintMonkey's practicality, we created the JungleGym dataset, which contains 100+ example Flask applications susceptible to web vulnerabilities from the Common Weakness Enumeration (CWE). When tested with web applications from JungleGym, TaintMonkey successfully detected all present vulnerabilities, demonstrating its readiness for real-life applications. Overall, TaintMonkey is an effective tool for detecting web application vulnerabilities in interpreted languages and requires minimal configuration effort from developers.**

I. INTRODUCTION

As the number of vulnerabilities and attack patterns that affect web applications exponentially increases, analysis of web applications becomes increasingly paramount in detecting and combating these vulnerabilities [1]. In 2024, over 40,000 new threats were discovered and added to the Common Vulnerability and Exception (CVE) database [1]. For example, the OWASP Top 10 ranks broken access control, server-side request forgery, and injection attacks as the leading web application security risks [2]. These vulnerabilities threaten

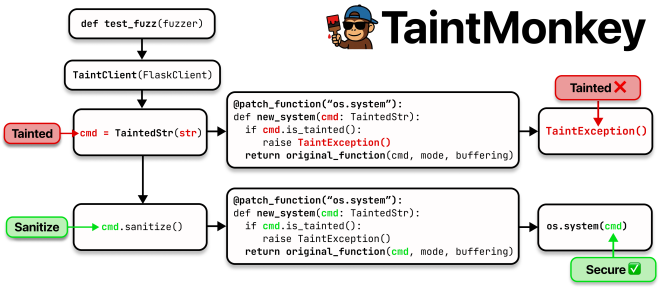

Fig. 1. TaintMonkey uses monkey patching to add taint analysis instrumentation to web applications. Taint analysis tracks the flow of potentially vulnerable data throughout a program

the confidentiality, integrity, and availability of web applications, creating significant concern for both corporations and consumers.

Efforts to build tools that automate the vulnerability detection process focus on two strategies: static and dynamic analysis. Static analysis provides a comprehensive way to detect weaknesses within web applications, but is often prone to false positives and theoretical outcomes improbable with real-world interpreters [3] [4]. Dynamic analysis, conversely, addresses these limitations by detecting vulnerabilities at runtime. In addition, the majority of web security research focuses on compiled languages (Java, C, C++) rather than interpreted languages (Python). This goal of this research is to lay the groundwork for practical vulnerability detection in interpreted languages by developing a dynamic analysis tool for Python web applications.

TaintMonkey, the dynamic analysis tool developed in this

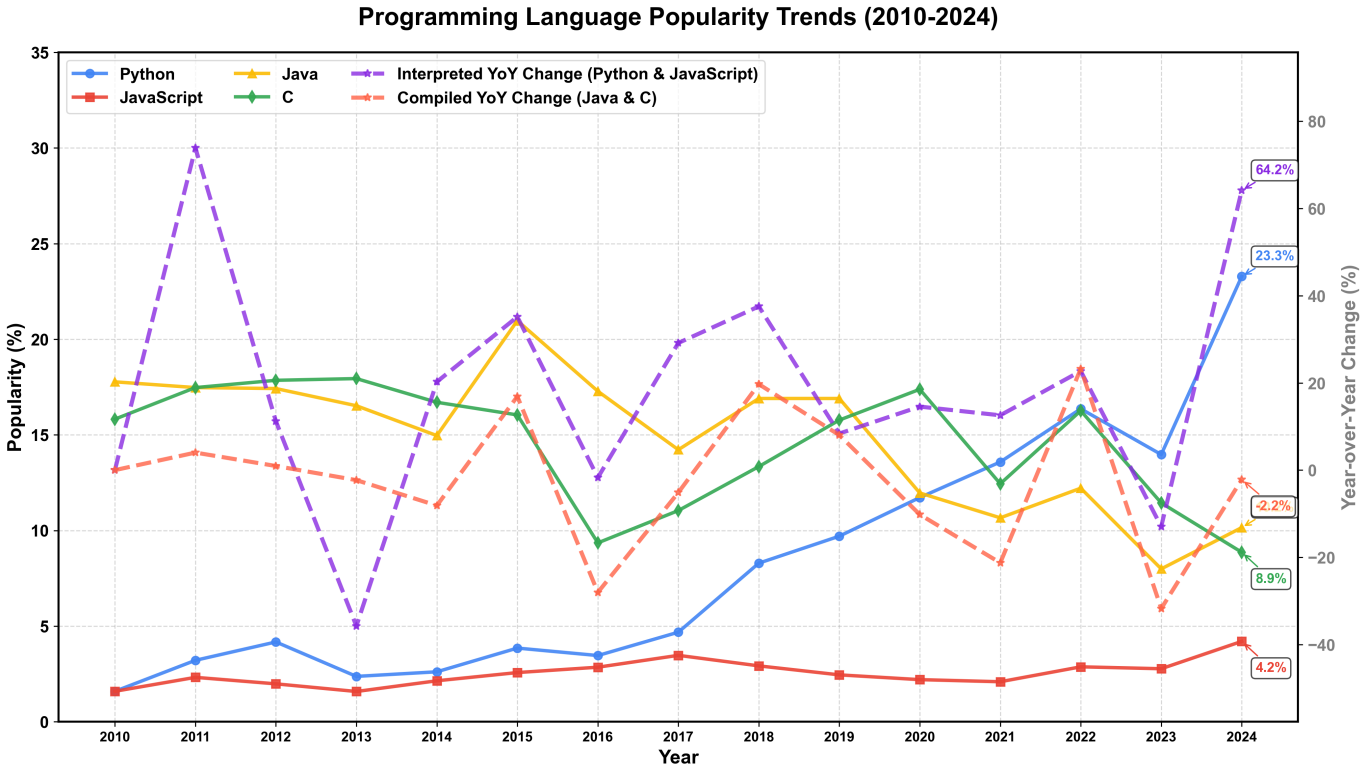

Fig. 2. Interpreted vs Compiled Languages Popularity Trends (2019-2024)

research, uses three primary techniques: taint analysis, monkey patching, and fuzzing. Taint analysis is the process of analyzing the data flow of a program by examining the output of a 'tainted' input. To perform taint analysis, TaintMonkey instruments a web application's sources to mark all inputs as tainted, sanitizers to untaint a certain input if passed, and sinks to check for tainted inputs. Dynamic instrumentation of these functions is accomplished through monkey patching. Monkey patching is the process by which the functionality of a class, function, or other object can be extended at runtime without modification of the source code [5]. For example, in Figure 1, the sink function "os.system" is monkey patched to check for tainted values. The interpreter will override the original function definition with the monkey patched definition shown.

TaintMonkey makes the assumption that the primary way adversaries interact with web servers is through requests. A fundamental part of how users communicate with web applications are HTTP requests, or messages sent from the client to the server, requesting a specific action (e.g. GET, POST, PUT). TaintMonkey aims to protect against adversaries that exploit such requests within Python web applications.

Because manual inputs struggle to cover all edge cases of a web application, TaintMonkey uses fuzzing to resolve this issue. In fuzzing, a fuzzer generates potentially harmful inputs, and a harness executes a target application using these inputs to expose vulnerabilities [6]. In particular, TaintMonkey performs semantic fuzzing, a technique that considers the behavioral effects of inputs, as well as information flow fuzzing, a technique that detects information flow through mutations of existing test inputs [7]. TaintMonkey fuzzes a given web application while monkey patching critical functions to add taint analysis instrumentation. During this fuzzing process, if a tainted/potentially harmful value is found to have reached the sink before first being sanitized, an exception is thrown to make the user aware of the vulnerability.

TaintMonkey presents a practical approach to vulnerability detection in Python web applications by addressing issues with current state-of-the-art vulnerability detection tools. Overall, TaintMonkey stands as a novel and scalable dynamic taint analysis tool that utilizes both fuzzing and monkey patching to detect interpreted-language-based web application vulnerabilities.

II. BACKGROUND

A. Program Analysis

Program analysis is the field concerned with understanding program semantics, or how a program behaves at runtime [8]. Although program analysis may be applied in areas besides cybersecurity, the scope of this paper focuses on how program analysis can be applied to detect vulnerabilities in web applications. There are two main types of program analysis: static and dynamic.

Static analysis involves extracting semantic properties of programs from source code and is lower in computational overhead, which requires less computer resources than dynamic analysis. Nonetheless, static analysis has theoretical

limitations. The Halting Problem, proposed by Alan Turing, explores the question of whether there is a general algorithm to determine if another program would halt in a finite time with given input. Ruled undecidable, no algorithm exists to determine if it may continue execution forever, or terminate eventually. Rice's Theorem extends the Halting Problem, stating that all non-trivial semantic properties (i.e. properties about a program's behavior that may hold true for some programs and false for others) are undecidable. Static analysis tools, such as Semgrep, CodeQL, and linters, fall short due to these limitations, forcing them to provide approximations of program behavior and leading to frequent false positives.

Dynamic analysis tools observe a program at runtime to determine its semantic properties. Dynamic analysis tools, such as fuzzers and debuggers, accomplish this by making use of instrumentation: modifying code to observe the program's runtime behavior and performance to a variety of test cases. As a result, they provide a more sound analysis of programs because they assess actual executions with concrete inputs. However, dynamic analysis may inhibit the performance of a program by adding computational overhead.

In developing program analysis tools, a tradeoff between soundness and completeness is often necessary. Soundness covers the idea of never making mistakes and proving only what is true. As a result, analysis with soundness has no false positives (false alarms). Conversely, completeness proves all true statements and catches every possible violation, leading to no false negatives. Static analysis follows completeness, leading to violations that may not always be correct. In other words, this lack of soundness means greater false positives.

B. Web Application Security

From 2024 to 2025, basic web application attacks increased to 12% of all security breaches [9]. Web application security aims to protect software on the internet from these attacks that expose sensitive information or break functionality. Web application security breaches are often the result of how the frontend, backend, database, and network of a web application are integrated [10]. Injection attacks like SQL injection and cross-site scripting (XSS) are some of the most prominent web vulnerabilities. These occur when web applications do not properly neutralize user input, allowing attackers to input malicious commands and force servers to execute them. Current state of the art (SOTA) practices to detect these vulnerabilities rely on either manual review or static analysis tooling like Semgrep and CodeQL, which often result in high false positive rates [11] [12]. Thus, as web application security becomes a growing concern, demand for precise and scalable detection tools is increasing.

C. Taint Analysis

Taint analysis is a program analysis method that tracks the flow of data by focusing on sources, sinks, and sanitizers. Tainting refers to the flagging of potentially malicious data coming from inputs known as sources. Sources are the origins of potentially dangerous data, typically from user input, and

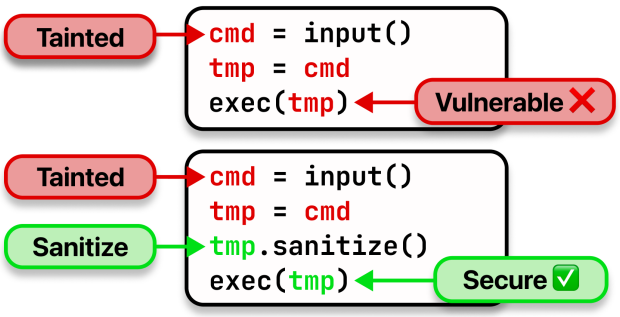

Fig. 3. Visualization of taint analysis. Vulnerable programs allow tainted data to flow from source to sink. Secure programs, sanitize tainted data before it can reach a sink

analysis tools taint data when they enter sources. Sinks are vulnerable functions or methods that can be reached by tainted data. For example, in Figure 3, the `input()` function would be referred to as the source, whereas the `exec(act)` function would be referred to as the sink. Taint propagation refers to the path tainted data takes to flow from source to sink, either via assignment or control flow statements. Sanitizer functions can cleanse tainted data, removing taint before it reaches a sink to make the data safe. However, if data reaches a sink while still tainted, the corresponding code block is flagged as a security concern.

There has been existing work with integrating taint analysis in interpreted languages like Perl and Ruby, although Python has lacked support [13] [14]. At the same time, however, these taint tracking mechanisms are rarely used because they are difficult to configure within large programs and lack granularity in how they taint data. In fact, Ruby 2.7 removed support for taint checking due to defects in the taint mechanism [15]. Additionally, there has been significant previous work exploring taint analysis techniques for detecting security vulnerabilities. The majority of such prior work has focused on static analysis techniques. For example, Tripp et al. presented TAJ, a static taint analysis tool written in Java that identifies web vulnerabilities [16]. It is meant to analyze industry-scale web applications, and can identify cross-site scripting, injection, malicious file execution, and information leakage attacks. However, TAJ struggles with soundness due to lack of context sensitivity, only supporting one level of call-string context in factory methods and in taint sources and sinks. Similarly, Semgrep and CodeQL support static taint tracking, relying on pattern recognition and resulting in a high false positive rate. For this reason, static analysis cannot track information flow as precisely as dynamic analysis techniques.

Dynamic taint analysis techniques for vulnerability detection have also been explored. One such example is Dytan, a customizable dynamic taint analysis framework allowing for data flow and control flow tainting while enabling users to define sources, sinks, and propagation policies [17]. Clause et al. demonstrated Dytan's efficacy through its ability to successfully detect overwrite injection and SQL injection attacks

with little implementation work. Dytan, however, was designed for binaries compiled from C/C++, which is less popular to use in web applications. Saoji et al. explored dynamic taint tracking in JavaScript to prevent web application attacks by automatically sanitizing strings [18]. Their API offers coarse-grained taint tracking, where an entire string is tainted and an exception is raised if this string reaches a sink. The API also supports precise, character-level taint tracking, which allows users to sanitize specific parts of strings without crashing the program. Despite these benefits, this API requires adding instrumentation to the source code. DynaPyt is a dynamic analysis tool with taint tracking features for Python programs [19]. DynaPyt includes an instrumenter that takes analysis programs written by developers as input, therefore allowing DynaPyt the ability to only observe program functionality of interest to the developer. During program execution, the instrumenter notifies the runtime engine whenever an event of interest occurs, which in turn calls analysis hooks that observe or manipulate the program's behavior. Although Eghbali and Pradel demonstrate that DynaPyt is faster than similar tools, it relies on static instrumentation, which requires significant effort from developers. While prior work has used dynamic taint analysis in interpreted languages, this work places a greater focus on detecting a wide breadth of web application vulnerabilities without modifying the source code.

D. Fuzzing

Fuzzing is a dynamic testing technique in which a target piece of software is tested with randomized data [20]. As shown by Güler et al., fuzzing is highly effective in web application security because it allows for target applications to be tested with payloads, or malicious inputs, that developers may overlook [21]. The primary goal of fuzzing is to increase code coverage, or how much of a program is tested. The greater the code coverage, the more paths in a program are explored, enabling developers to identify more bugs. The fuzzer interfaces with a harness, which retrieves test inputs from the fuzzer and passes them to the target. There are several types of fuzzers, and TaintMonkey includes implementation of three strategies: a dictionary, mutation, and grammar-based fuzzers.

1) Dictionary-Based Fuzzing: In dictionary-based fuzzing, the fuzzer is provided a dictionary text file containing possible inputs [22]. The fuzzer then samples inputs from the dictionary at random to generate inputs.

2) Mutation-Based Fuzzing: Similar to dictionary-based fuzzers, mutation-based fuzzers require users to provide an initial corpus of valid inputs. Mutation-based fuzzers generate new test cases by making random alterations, or mutations, to the inputs. For example, with strings, possible mutations include inserting characters, deleting characters, reordering characters, or flipping character bits. The goal of mutation-based fuzzing is to increase code coverage by exploring new execution paths using inputs derived from known, valid inputs.

3) Grammar-Based Fuzzing: Grammar-based fuzzing allows users to develop structured test inputs without manually writing them [7]. A user must first provide a grammar, which specifies the input syntax. For example, a phone number grammar would specify the syntax for a phone number. Each line of a grammar consists of a symbol, representing a component of the generated input, followed by an expansion rule that details how that symbol can be replaced with other symbols. A symbol can either be nonterminal, meaning it can be expanded further, or terminal, meaning it cannot be replaced any further. A grammar begins with the expansion rule for a symbol that represents the whole input being generated. When executed, the fuzzer traverses the grammar recursively to construct an input.

```
### Grammar-Based Fuzzing Example

US_PHONE_GRAMMAR: Grammar = {
  "<phone-number>": ["(<area>)<exch>-<line>"],
  "<area>": ["<lead-dig><dig><dig>"],
  "<exch>": ["<lead-dig><dig><dig>"],
  "<line>": ["<dig><dig><dig>"],
  "<lead-dig>": ["0", "1", "2", "3", "4",
  ↪  "5"],
  "<dig>": ["0", "1", "2", "3", "4", "5", "6"]
}
```

E. Monkey Patching

Monkey patching is a feature of many interpreted languages, such as JavaScript, Ruby, and Python, which allows a program to be extended without directly modifying the source code [5]. Monkey patching is commonly used in unit testing and mocking, such as with the `pytest` and `unittest.mock` libraries in Python [23] [24]. Because monkey patching allows for instrumentation without altering source code, developers can conveniently patch functions to perform taint analysis on Python web applications.

```
### Monkey Patching Example

import random

def randint(a: int, b: int) -> int:
    return 42

# Monkey patch the randint method
random.randint = randint

# Always prints 42
for _ in range(10):
    print(random.randint(0, 10))
```

III. SYSTEM DESIGN

This section explores the components behind TaintMonkey: the fuzzer, harness, executor, and observer. Developers specify the type of fuzzer (dictionary, mutation, or grammar), which either pulls test cases from the corpus, mutates a given set of test cases, or generates test cases according to a grammar. Once initializing the web application using Flask's Web Server Gateway Interface (WSGI) test client, TaintMonkey uses a

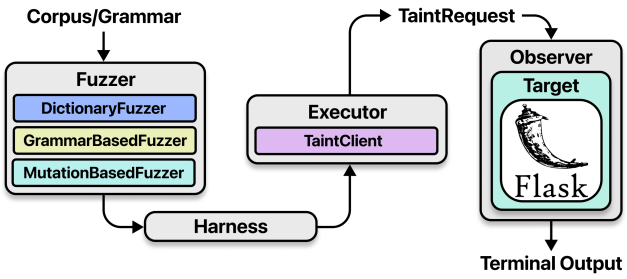

Fig. 4. An Overview of TaintMonkey. The harness takes inputs generated by the fuzzer and passes them to the executor, which makes a request with the provided inputs. The observer then performs taint analysis as the target Flask application runs.

fuzzing harness to hit endpoints and specific routes with potentially harmful inputs. Flask's WSGI's test client also allows tainted requests to be sent from the client into the target, a Flask web application. As requests are sent, the observer records potential vulnerabilities by analyzing the application's data flow. These potential vulnerabilities are outputted in the terminal to a web developer to be analyzed and fixed.

A. Fuzzer

TaintMonkey implements an interface for a fuzzer. As seen in Figure 4, the fuzzer begins by loading a set of input payloads, also known as test inputs, from the corpus or grammar. In TaintMonkey, the fuzzer is created to expose vulnerabilities in a developer's Flask web application by connecting to the application via the harness. The fuzzing harness is necessary as the fuzzer provides and generates the test cases, while the harness implements those test cases into the web application's sources. TaintMonkey provides an interface with an abstract base class that allows for the implementation of three types of fuzzers:

- **DictionaryFuzzer**: Inheriting methods and attributes from Fuzzer, this subclass randomizes the order of fuzz inputs passed onto the `DictionaryFuzzer` from the corpus file. Additionally, the child class gets a Flask test client, allowing them to send requests to the Flask app being tested. The corpus file, `dictionary.txt`, is used by the `DictionaryFuzzer` to pull payloads which are then implemented by the fuzzing harness.
- **GrammarBasedFuzzer**: TaintMonkey includes a grammar-based fuzzer that generates valid JSON inputs. The fuzzer script was generated by the `Grammarinator` fuzzer generator, which was inputted with a JSON grammar in the ANTLRv4 language [25] [26]. TaintMonkey's grammar-based fuzzer is also customizable, allowing developers to adjust the fuzzer to various data types, such as dictionaries, lists, strings, integers, etc. Motivated by Olsthoorn et al.'s work on adding mutations to grammar-based fuzzing, TaintMonkey's grammar-based fuzzer allows developers to add a list of keys, which are sampled at random and inserted into the JSON input [27]. For example, if a developer is testing a login page, they can provide the grammar-based fuzzer with the "username" and "password" keys.
- **MutationBasedFuzzer**: TaintMonkey's mutation-based fuzzer takes a seed corpus of valid string inputs for a given vulnerability type, randomly samples from it, and performs a series of mutations on each input. These mutations were taken from the Frelatage fuzzing library, and include flipping bits, byte substitution, inserting characters, deleting characters, swapping characters, duplicating substrings, repeating substrings, and deleting substrings [28]. Developers can designate the minimum and maximum mutations they want the fuzzer to perform, which acts as a range the fuzzer randomly chooses from. Developers can also customize the minimum and maximum length of the generated inputs.

B. Harness

Fuzzing harnesses are integral to the execution of fuzzers. By itself, the fuzzer solely feeds large amounts of data by sampling inputs from the corpus file/grammar. As a result, the fuzzer relies on a fuzzing harness function that defines how payloads are sent to the Flask web application. This process involves receiving inputs from the fuzzer and running them through the application. Security engineers specify which endpoints to hit, such as the Flask route that the fuzzer is targeting. According to the endpoints, inputs are formatted in order to send test payloads. In writing a fuzzing harness for TaintMonkey, identifying the target is essential to tracing the taint source. In this case, the target Flask application must be imported. Next, the plugin runs a test function (e.g. `test_fuzz`) using `pytest`, which allows the fuzzer to be passed and to perform taint tracking. After that, the fuzzing context is established through the Flask test client `TaintClient`. Once the fuzzing harness is set up, the application marks inputs as tainted and proceeds to detect tainted inputs in sinks. A `TaintException` will be thrown if a tainted input is found to have reached the application's sink.

```
### Fuzzing Harness Example

def test_fuzz(taintmonkey):
  fuzzer = taintmonkey.get_fuzzer()
  with fuzzer.get_context() as \
    (client, get_input):
      for inp in get_input():
        client.get(f"/insecure?file={inp}")
```

C. Executor

The executor serves as the component that runs the Flask web application with the fuzzed input and executes the taint-tracking logic. TaintMonkey uses Flask's WSGI test client to run the web application in a controlled environment while ensuring that inputs are tainted correctly and propagate throughout the program. Flask is implemented using `Werkzeug`, a

comprehensive WSGI application library that provides a client to simulate requests to a WSGI application without having to start a server. Within this client, there are methods for making different types of requests and managing state [29]. Specifically, there is `TaintClient`, a custom subclass of `FlaskClient` created for TaintMonkey. It takes advantage of Python's duck typing to clobber the `Werkzeug` test client's request handler `.open()` function for tainting. By overriding the `.open()` method, every request sent will now automatically be marked as tainted.

Taint-tracking logic follows the idea of dynamic taint propagation, the process of automatically tracking how tainted data spreads throughout the program as it executes. By monitoring the target program as it runs, Dynamic taint propagation associates a "taint marker" to user input. In TaintMonkey, the taint marker TaintedStr propagates through the program until it reaches a sink, in which the monkey-patched function may raise a `TaintException` and report the vulnerability [30].

TaintMonkey uses monkey patching as an instrumentation technique to implement such taint-tracking logic with strings. Focusing on server-side programs and dynamic web pages such as Flask, potentially unsafe input strings may compromise security and vulnerability protection. As observed by Minamide, strings represent potentially unsafe inputs and therefore may be vulnerable for web pages [31]. Resulting vulnerabilities such as cross-site scripting (XSS) require sanitization to ensure safety when embedding into a web page. Moreover, real-world situations typically involve strings from sources that are then manipulated to form other strings such as queries and scripts. Thus, strings derived from tainted strings also need to be marked as tainted to maintain taint propagation [32].

D. Observer

The observer is the component that captures feedback from the fuzzer, recording possible vulnerabilities within Python Flask web applications. The observer's role in TaintMonkey encompasses detecting when tainted input reaches a dangerous sink and then calling a `TaintException` if a potential vulnerability is found through patching. `Pytest`, a framework that features functional and readable testing for applications, runs the harness, thus simulating attacks. Afterwards, `pytest` outputs the results and flags each test depending on whether an exception is raised [33].

IV. EXPERIMENTAL PROCEDURES

The goal of this experiment was to evaluate TaintMonkey's ability to detect vulnerabilities. To accomplish this, TaintMonkey was tested across controlled web applications representing common vulnerability patterns. The results were then utilized as feedback to re-evaluate and improve the TaintMonkey library.

A. Dataset

To create a consistent and efficient evaluation environment, several web applications were developed, simulating common vulnerabilities with paired endpoint flow. Each pair consisted of an insecure implementation and a corresponding secure variant that included an appropriate sanitizer for the specific Common Web Enumeration (CWE) vulnerability.

- **Insecure Endpoint**: An implementation of a web service interface that lacks proper security controls and input validation, making it vulnerable to attacks.
- **Secure Endpoint**: An implementation of a web service interface that incorporates comprehensive security controls, proper input validation, and follows security practices to protect against various attack vectors.

This structure isolates the effects of taint tracking by preserving application logic while varying only the presence of security controls. Each application exposed one or more HTTP endpoints, which simulates real-world web functionalities. These endpoints accept user input via a variety of request objects (JSON, Form Data, etc.), which was then subjected to a sequence of transformation steps such as string concatenations and input encoding. After processing, the transformed input was routed to the user's designated sink function (e.g. file write, database call, or command execution) representing potential vulnerability points in web applications. The sink functions were chosen based on the specific CWE vulnerability pattern being tested. This approach allows for analysis of TaintMonkey's ability to detect potentially malicious flows in the insecure version and avoid false positives in the secure counterpart.

The following vulnerabilities were represented in the test suite database:

- **CWE-73: External Control of File Name or Path (External File Control).** Occurs when an application allows all user inputs to be directly used in functions to open, create, delete, or modify files, allowing attackers to gain full control of file names or paths used by the application [34].
- **CWE-78: Improper Neutralization of Special Elements in OS Command (OS Command Injection).** Occurs when web applications improperly sanitize user inputs, allowing a user to pass unsafe data to a system shell in the form of an OS command. Depending on the OS command being executed, an attacker can chain their payloads to the command or completely replace the intended command strings with their own [35].
- **CWE-79: Cross-site Scripting (XSS).** Allows attackers to inject malicious client-side scripts into web pages viewed by other users. In such web applications, accepting unsanitized, malicious user input can lead to session credential theft, hijacking systems, or delivering unwanted actions to victims [36].
- **CWE-89: Improper Neutralization of Special Elements in SQL Command (SQL Injection).** Enables attackers to manipulate database queries by injecting malicious SQL syntax that exposes or modifies sensitive information from the database. This can potentially also lead to modification and destruction of database contents

when compiled [37].

- **CWE-94: Improper Control of Generation of Code (Code Injection).** Occurs when user input is directly used to generate and execute code for Python's eval(), exec(), or shell eval without proper validation or sanitization [38].
- **CWE-200: Exposure of Sensitive Information to an Unauthorized Actor (Sensitive Data Exposure)** Occurs when an application has a protection gap leading to accidental revealing of sensitive information such as credentials, user data, or system details that should otherwise remain confidential [39].
- **CWE-306: Missing Authentication for Critical Function (Missing Function Auth).** Occurs when an application does not properly verify if a user has the required permissions to access functionality or confidential resources, allowing unauthorized privilege escalation [40].
- **CWE-352: Cross-Site Request Forgery (CSRF).** Forces authenticated users to execute unwanted malicious commands on web applications where they are currently authenticated. Attackers leverage the trust a site has in a user's browser software to exploit the user's active session, submitting an unintentional state change request to the web server [41].
- **CWE-434: Unrestricted Upload of File with Dangerous Type.** Allows for the upload or transfer of files of a dangerous type that are automatically processed and accessed within a web application's environment [42].
- **CWE-918: Server-Side Request Forgery (SSRF).** Occurs when a server-side application makes a request to an unintended source allowing the attacker to trick the server into sending HTTP requests on their behalf (typically caused by unsanitized user input) [43].
- **CWE-938: URL Redirection to Untrusted Site (Open Redirect).)** Attackers redirect users to malicious websites by manipulating URL parameters. These attacks often appear legitimate as they originate from trusted domains and potentially lead to unwanted command execution on the user's browser [44].

Each test case was stored in a standardized format: a standalone directory containing an `app.py` file with the Flask application logic and a `requirements.txt` file to ensure reproducibility and scalability. These test cases were compiled into a dataset named JungleGym.

B. Plugin Development

After creation of the JungleGym dataset, a series of plugins to test for and identify vulnerabilities within JungleGym were developed. To perform dynamic taint analysis, each plugin instrumented the web application's sources, sinks, and sanitizers via the monkey patching function `patch_function`. Sources were patched to taint all inputs, sanitizers were patched to untaint the inputs, and sinks were patched to check for tainted data. Each plugin defines its target web application, as well as the specific fuzzer (dictionary, grammar, or mutation) to trigger each vulnerability class and potential vector route. These fuzzers generated payloads by randomly generating test cases according to a dedicated dictionary, mutation on a dictionary, or with a given grammar. Payloads were then applied using TaintMonkey's `test_fuzz` function, which systematically injected these payloads into test endpoints during evaluation.

TABLE I
OVERVIEW OF TAINTMONKEY DATASET (JUNGLEGYM)

CWE	#	Libraries
73 External File Control	12	Werkzeug
78 OS Command Injection	8	blinker, click, itsdangerous, Jinja2, MarkupSafe, Werkzeug
79 XSS	10	bleach, html-sanitizer, lxml, MarkupSafe
89 SQL Injection	9	SQLAlchemy, Flask-SQLAlchemy, requests, werkzeug
94 Code Injection	10	blinker, click, itsdangerous, Jinja2, MarkupSafe, Werkzeug
200 Sensitive Data Exposure	10	Faker
306 Missing Function Auth	11	flask_mail, Flask_Login
352 CSRF	10	flask_wtf, werkzeug, itsdangerous, flask_seasurf
434 Unrestricted File Upload	10	blinker, click, itsdangerous, Jinja2, MarkupSafe, uuid, python_magic, Requests
918 SSRF	10	furl, Requests, yarl
938 Open Redirect	10	furl, Requests, yarl

C. Testing Framework

TaintMonkey tests reference applications through a combination of manual and automated testing with plugins from the dataset. TaintMonkey's testing framework involves creating a `test_client` that is used as a reference application to test out each security plugin. Manual testing is implemented first, calling a `test_taint_exception()` and `test_no_taint_exception()` that load specific test inputs (e.g. `<script>alert('XSS')</script>`). The `test_taint_exception()` method ensures that a TaintException is raised when a request is made to the insecure route. Conversely, `test_no_taint_exception()` follows the secure route, and verifies that no `TaintException` is raised as a result of proper sanitization. Afterwards, automated testing leverages the fuzzing harness to generate test inputs that detect vulnerabilities. Using the harness, `pytest` runs multiple tests at an insecure route to raise a possible `TaintException` and output a potential vulnerability.

D. Benchmarking

To evaluate the TaintMonkey framework, malicious payloads are first inputted to JungleGym's insecure and secure implementations. In the insecure route, the user's payload was passed directly to an absent or malfunctioning sink function (e.g., `exec(cmd)`, `os.system()`) without validation, allowing us to observe true positives and any false negatives. In contrast, the secure implementation applied appropriate payload sanitization based on the test case's parameters before the input reached the sink, enabling proper route security and evaluation of false positives and true negatives. User inputs

TABLE II
COMPARISON OF FEATURES BETWEEN TAINTMONKEY AND OTHER TAINT ANALYSIS TOOLS

Attributes	TaintMonkey	DynaPyt	Dytan	TAJ	TaintModePythonLibrary
Targets Interpreted Languages	✓	✓	✗	✗	✓
Targets Web Apps	✓	✓	✗	✓	✓
Dynamic Analysis	✓	✓	✓	✗	✓
Monkey Patching	✓	✗	✗	✗	✗
Integrated Fuzzer	✓	✗	✗	✗	✗

in the secure route were successfully marked as tainted and raised a `TaintException` if not passed through a sanitizer. Sanitizers within secure routes cleaned and untainted payloads prior to reaching the sink, allowing for data to pass without raising a `TaintException`. The insecure route, on the other hand, successfully did not raise a `TaintException`, allowing for the passage of potentially dangerous input. With this pair, TaintMonkey then compared the insecure and secure paths to improve accuracy, demonstrating TaintMonkey's ability to distinguish between insecure and secure code paths.

Based on these findings, test cases and sanitizer logic were refined to fix uncovered security gaps. This iterative process ensures TaintMonkey's detection ability to reduce false positive output cases.

V. RESULTS

TaintMonkey, a library, was created as a tool to help expose vulnerabilities in web applications. To gauge its effectiveness, it has been both quantitatively and qualitatively analyzed against possible test cases and other SOTA. This section will display the related findings.

A. Objective Performance Evaluation

TaintMonkey was manually tested using a custom dataset, JungleGym, with 100+ test cases spanning across 11 different CWEs related to the OWASP Top 10 [45]. TaintMonkey was able to address every test case type in JungleGym via the development of various custom plugins. The plugin uses the TaintMonkey framework to track tainted data, generate and send test inputs to specific routes, and call `pytest` to output potential vulnerabilities. During the development of test cases and plugins, TaintMonkey's components were progressively refined to provide improved robustness, functionality, and detection with greater efficacy.

B. Comparison to SOTA

To gauge TaintMonkey's performance, its attributes were compared against those of other analysis tools found in literature. Specifically, information was collected on four tools: DynaPyt, Dytan, TAJ, and TaintModePythonLibrary (TMPL). The documented comparisons are depicted in the table shown in Table II.

DynaPyt and TMPL are dynamic analysis tools for general Python application purposes that use dynamic taint analysis similar to TaintMonkey [19] [46]. Monkey patching is not used in DynaPyt nor TMPL. Instead, DynaPyt utilizes a static code review before runtime to insert hooks into imports, manipulating the code as opposed to the objects at runtime. Only after code modification does it perform dynamic analysis. Meanwhile, TMPL uses decorators to mark sources, sanitizers, and sinks, directly modifying source code. However, TaintMonkey taints values using monkey patching, which does not modify any source code. This makes TaintMonkey a simplistic and safe tool to use, as the testing environment is completely separate from the source code. Moreover, Dytan and TAJ do not support interpreted languages, such as Python. Dytan focuses on x86 binary executables while TAJ is for Java, a compiled language [17] [16]. Monkey patching is primarily an attribute of interpreted languages; thus, neither Dytan nor TAJ supports this feature. Additionally, Dytan does not target web applications specifically, unlike TaintMonkey, which has been created with the specific intent to target web vulnerabilities. Likewise, while TAJ targets web applications like TaintMonkey, its static taint analysis approach, which, as stated before, has limitations due to the Halting Problem and Rice's Theorem. Furthermore, although TMPL targets web applications built with interpreted languages and uses dynamic analysis, it does not contain an integrated fuzzer, nor does it utilize monkey patching [46]. Conversely, TaintMonkey integrates fuzzers to test all potentially dangerous edge cases for a given web application.

Comparing TaintMonkey to other SOTA tools, it's clear that TaintMonkey accomplished the goal of being a robust tool that encapsulates all five attributes shown in the table: targets interpreted languages, targets web applications, uses dynamic analysis, has monkey patching support, and includes an integrated fuzzer. TaintMonkey takes a unique approach to program analysis that differentiates it from other programs, solidifying itself in its niche.

C. Limitations

TaintMonkey has certain limitations concerning its performance at identifying vulnerabilities. Taint analysis observes data flow from source to sink; however, if it is unclear how to

hook two different outcomes to a source, taint analysis may fail. For example, see below. The user input (the source) is stored in a tainted string called `foo` and is marked as tainted. The value of `foo` determines whether the variable `safe` is true or false. Further down the code, there is a function named `danger()` that, if executed, will result in a vulnerability. This dangerous function executes if `safe` is true. While the value of `foo` leads to whether this failure occurs, it is not directly involved in the execution of the sink. Data flow analysis cannot catch this without large modifications to the source code, which both defeats the purpose of TaintMonkey's monkey patching features and implies that the developer is aware of the possibility of vulnerability in the first place, which is not always true and would make TaintMonkey's analysis redundant. However, this type of flow-sensitive taint is uncommon in web applications. Direct injection vulnerabilities (what dynamic taint analysis excels at catching), on the other hand, are much more prevalent in these applications.

```
### Example illustrating TaintMonkey's
### limitations in monitoring
### control flow tainting

foo = input()
if foo == "bar"
    safe = True
else:
    safe = False
if safe:
    danger()
```

Additionally, situations with identical data flow characteristics across test instances are not optimal for taint analysis either. One example of this is a session cookie, which is sent with every request to the client application; it is constantly being renewed. The flow of a malicious, injected cookie is identical to that of a benign cookie assigned by the server. During each request, the client retrieves the cookie from the session (what would be the source) and uses it to verify the identity of the user (this function would be the sink). Sanitizing a cookie does not fit into this framework. If TaintMonkey were to be utilized, it would mirror the behavior of a check/verifier instead. This holds similar problems to the hooking issue described earlier, where the developer would most likely be aware of the possible vulnerability in the first place, thus making the analysis redundant. Other methods than taint analysis are better suited for problems like this. TaintMonkey's focus on dynamic taint analysis is optimized for injection-based vulnerabilities.

VI. CONCLUSION

A. Significance of Findings

TaintMonkey, unlike other analysis tools, uses monkey patching to instrument a target program, removing the need for source code modification. Monkey patching, a relatively easy technique to implement, stands as a practical solution to utilize program analysis for interpreted languages. TaintMonkey also includes dictionary, mutation, and grammar-based fuzzers, which help developers thoroughly test execution paths in their program. Using these techniques in the custom dataset JungleGym, which contains 100+ unique test cases, TaintMonkey has proven that it can detect a wide variety of vulnerabilities. Overall, TaintMonkey stands to be a robust dynamic taint analysis tool for detecting vulnerabilities in Python Flask web applications.

B. Future Work

In the future, TaintMonkey can be augmented by enabling it to track taint from sources other than just Flask requests. For example, in some persistent XSS attacks, malicious scripts are stored in databases and later read and executed by a server. In TaintMonkey's current implementation, there is no way to preserve taint markings while data is stored in a database. A plan to tackle this limitation involves monkey patching database APIs and manually propagating taint. This approach can then be tested with fuzzing by using `pytest` to mock database behavior.

In addition, support can be added in TaintMonkey for more tainted data types, not just tainted strings. Currently, TaintMonkey does not correctly taint built-in data types such as string literals and integers. In order to increase the scope of its practical applications, TaintMonkey should effectively monkey patch a broad range of data types.

Another application of TaintMonkey's approach to dynamic taint analysis with monkey patching is the issue of privacy leakage. Potential leaks involve the exposure of personally identifiable information (PII) due to improper logging and storage of data. Certain legal entities, such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA) strictly enforce the right to be forgotten, an individual's right to have their personal data deleted from any web application's database [47] [48]. Infractions of these regulations by the accidental leakage of personal data can result in steep financial penalties towards corporations. As shown by the CWE 200-Sensitive Data Exposure test cases and plugin, the application of TaintMonkey and its techniques can be used to detect privacy exposure, which not only protects corporations from potential liability with sensitive data handling, but also protects consumers who share their PII with these corporations from accidental PII exposure.

A common technique used in industry is snapshot fuzzing. Snapshot fuzzing is the process of taking a "snapshot" of the target program right before the test input is processed, capturing the memory. A snapshot allows for fuzzing to occur in an emulated environment [49]. If a `TaintException` were to be thrown, the fuzzer has the ability to reset back and further test with additional payloads. Employing snapshot fuzzing has numerous advantages that may be applicable to TaintMonkey. Snapshot fuzzing means that the same test input must return the same result, and allows for reproducible test cases. Above all, snapshot fuzzing is fast, as it doesn't require setting up the web application for each test run. However, there is difficulty with implementing this with monkey patching

due to the desire to implement lightweight instrumentation. Snapshot fuzzing involves control over a target application's state, while monkey patching only offers this control on specific functions. Thus, snapshot fuzzing alongside monkey patching results in inconsistency and may not properly apply the patch without modifying source code.

We have linked our GitHub repository for this project here: https://github.com/bliutech/TaintMonkey.

VII. Acknowledgments

The authors of this paper gratefully acknowledge the following: Residential Teaching Assistant and Research Coordinator Anusha Iyer for her invaluable assistance and guidance; Project Mentor Benson Liu for his valuable knowledge of computer science and hands-on involvement; Dean Jean Patrick Antoine, the Associate Director of GSET for his management and guidance; Rutgers University, Rutgers School of Engineering, and the State of New Jersey for the chance to advance knowledge, explore engineering, and open up new opportunities; Lockheed Martin and Commercial Metals Company (CMC) for funding of our scientific endeavours; and lastly NJ GSET Alumni, for their continued participation and support.

References

[1] CVE Program, "Published CVE Records," https://www.cve.org/About/Metrics, 2025, Accessed: 2025-07-19.

[2] OWASP Foundation, "OWASP Top Ten Web Application Security Risks," https://owasp.org/www-project-top-ten/, 2025, accessed: July 19, 2025.

[3] J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of javascript web applications in the wild," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 61–70.

[4] P. Thomson, "Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity." *Queue*, vol. 19, no. 4, pp. 29–41, 2021.

[5] J. Hunt, "Monkey patching," in *A Beginners Guide to Python 3 Programming*. Springer, 2023, pp. 487–490.

[6] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2020.

[7] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "About this book," in *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024, retrieved 2024-07-01 16:50:18+02:00. [Online]. Available: https://www.fuzzingbook.org/html/index.html

[8] X. Rival and K. Yi, *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[9] Verizon Statistic, "2025 Data Breach Investigations Report," https://www.verizon.com/business/resources/reports/dbir/, 2025, Accessed: 2025-07-19.

[10] S. Kumar, R. Mahajan, N. Kumar, and S. K. Khatri, "A study on web application security and detecting security vulnerabilities," in *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*. IEEE, 2017, pp. 451–455.

[11] Semgrep, "Taint analysis — Semgrep," https://semgrep.dev/docs/writing-rules/data-flow/taint-mode, 2025, Accessed: 2025-07-19.

[12] CodeQL, "CodeQL," https://codeql.github.com/, 2025, Accessed: 2025-07-19.

[13] Perldoc Browser, "Perl Documentation," https://perldoc.perl.org/, 2025, Accessed: 2025-07-19.

[14] D. Thomas, A. Hunt, and C. Fowler, *Programming Ruby 1.9 & 2.0: the pragmatic programmers' guide*. Pragmatic Bookshelf, 2013.

[15] Ruby Programming Language, "NEWS-2.7.0 - Documentation for Ruby 3.4," https://docs.ruby-lang.org/en/3.4/NEWS/NEWS-2_7_0.html, 2025, Accessed: 2025-07-19.

[16] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87–97, 2009.

[17] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 196–206.

[18] T. Saoji, T. H. Austin, and C. Flanagan, "Using precise taint tracking for auto-sanitization," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017, pp. 15–24.

[19] A. Eghbali and M. Pradel, "Dynapyt: a dynamic analysis framework for python," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 760–771.

[20] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[21] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective fuzzing of web applications for {Server-Side} vulnerabilities," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4765–4782.

[22] Testing Handbook, "Fuzzing dictionary," https://docs.ruby-lang.org/en/3.4/NEWS/NEWS-2_7_0.html, 2025, Accessed: 2025-07-19.

[23] Pytest Documentation, "How to monkeypatch/mock modules and environments," docs.pytest.org.https://docs.pytest.org/en/stable/how-to/monkeypatch.html, 2025, Accessed: 2025-07-19.

[24] Python, "How to monkeypatch/mock modules and environments," docs.pytest.org.https://docs.pytest.org/en/stable/how-to/monkeypatch.html, 2025, Accessed: 2025-07-19.

[25] renatahodovan, "GitHub - renatahodovan/grammarinator: ANTLR v4 grammar-based test generator," https://github.com/renatahodovan/grammarinator, 2025, Accessed: 2025-07-19.

[26] Antlr Project, "Grammars-v4," https://github.com/antlr/grammars-v4, 2025, Accessed: 2025-07-19.

[27] M. Olsthoorn, A. van Deursen, and A. Panichella, "Generating highly-structured input data by combining search-based testing and grammar-based fuzzing," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1224–1228.

[28] Rog3rSm1th, "Frelatage," https://github.com/Rog3rSm1th/frelatage/tree/main, 2025, Accessed: 2025-07-23.

[29] Werkzeug Documentation, "Testing WSGI Applications," https://werkzeug.palletsprojects.com/en/stable/test/, 2025, Accessed: 2025-07-23.

[30] Brian, Chess and Jacob, West, "Dynamic taint propagation: Finding vulnerabilities without attacking," https://www.sciencedirect.com/science/article/pii/S1363412708000058, 2025, Accessed: 2025-07-23.

[31] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th international conference on World Wide Web*, 2005, pp. 432–441.

[32] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005, pp. 9–pp.

[33] Pytest Documentation, "pytest: helps you write better programs — pytest documentation)," https://docs.pytest.org/en/stable/, 2025, Accessed: 2025-07-19.

[34] CWE, "CWE-73: External Control of File Name or Path (4.10))," https://cwe.mitre.org/data/definitions/73.html, 2025, Accessed: 2025-07-19.

[35] ——, "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')(4.2)," https://cwe.mitre.org/data/definitions/73.html, 2025, Accessed: 2025-07-19.

[36] ——, "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')(4.2)," https://cwe.mitre.org/data/definitions/78.html, 2025, Accessed: 2025-07-19.

[37] ——, "CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')(4.1)," https://cwe.mitre.org/data/definitions/79.html, 2025, Accessed: 2025-07-19.

[38] ——, "CWE-94: Improper Control of Generation of Code ('Code Injection')(4.4)," https://cwe.mitre.org/data/definitions/94.html, 2025, Accessed: 2025-07-19.

[39] ——, "CWE-200: Exposure of Sensitive Information to an Unauthorized Actor(4.1)," https://cwe.mitre.org/data/definitions/200.html, 2025, Accessed: 2025-07-19.

[40] ——, "CWE-306: Missing Authentication for Critical Function(4.6)," https://cwe.mitre.org/data/definitions/306.html, 2025, Accessed: 2025-07-19.

[41] ——, "CWE-352: Cross-Site Request Forgery(CSRF)(3.4.1)," https://cwe.mitre.org/data/definitions/352.html, 2025, Accessed: 2025-07-19.
[42] ——, "CWE-434: Unrestricted Upload of File with Dangerous Type," https://cwe.mitre.org/data/definitions/434.html, 2025, Accessed: 2025-07-19.
[43] ——, "CWE-918: Server-Side Request Forgery(SSRF)(4.4)," https://cwe.mitre.org/data/definitions/918.html, 2025, Accessed: 2025-07-19.
[44] ——, "CWE-601: URL Redirection to Untrusted Site('Open Redirect')(4.5)," https://cwe.mitre.org/data/definitions/601.html, 2025, Accessed: 2025-07-19.
[45] OWASP, "OWASP Top Ten," https://owasp.org/www-project-top-ten/, 2025, Accessed: 2025-07-19.
[46] J. J. Conti and A. Russo, "A taint mode for python via a library," in *Nordic Conference on Secure IT Systems*. Springer, 2010, pp. 210–222.
[47] Intersoft Consulting, "General Data Protection Regulation," https://gdpr-info.eu/, 2025, Accessed: 2025-07-23.
[48] Rob Bonta, "California Consuemr Privacy Act (CCPA)," https://oag.ca.gov/privacy/ccpa, 2025, Accessed: 2025-07-23.
[49] Testing Handbook, "Snapshot Fuzzing," https://appsec.guide/docs/fuzzing/snapshot-fuzzing/, 2025, Accessed: 2025-07-23.