NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSC THESIS

Hybrid Taint Analysis for Vulnerability Detection of XSS & SQL Injection in Django

Georgios E. Koursiounis

Supervisors: **Alex Delis,** Professor NKUA

ATHENS

DECEMBER 2020

ΕΘΝΙΚΟ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υβριδική Ανάλυση για Ανίχνευση XSS & SQL Injection στο Django

Γεώργιος Ε. Κουρσιούνης

Επιβλέποντες: **Αλέξιος Δελής**, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2020

BSC THESIS

Hybrid Taint Analysis for Vulnerability Detection of XSS & SQL Injection in Django

Georgios E. Koursiounis
S.N.: 1115201600077

SUPERVISORS: **Alex Delis,** Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υβριδική Ανάλυση για Ανίχνευση XSS & SQL Injection στο Django

Γεώργιος Ε. Κουρσιούνης
Α.Μ.: 1115201600077

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Αλέξιος Δελής**, Καθηγητής ΕΚΠΑ

ABSTRACT

In this thesis, we implement an execution monitor which combines hybrid (dynamic & static) taint analysis and server-side parsing in order to discover context-sensitive XSS flaws in template rendering, context-insensitive XSS flaws in simple HTTP responses and potential SQL injection in raw SQL queries in Django web applications.

It is observed that many industry solutions implement context-insensitive auto-sanitization as a main defense strategy. However, this provides a false sense of security, since untrusted data need to be sanitized differently based on their browser context. Therefore, the necessity for control over context-sensitive flaws is strikingly evident. Moreover, Django has no auto-sanitization policy regarding simple HTTP responses and, consequently, it is indisputable that attention has to be given also to simple HTTP responses. Last, raw SQL queries are often used when Django Object-Relational Mapper (ORM) is proved not enough. Thus, this might pose a security risk if not handled correctly.

We provide an analysis tool via a library entirely written in Python based on the approach presented by Conti & Russo [2] and later adopted by Steinhauser & Tůma [1]. No modifications in the interpreter are needed. To do so, we conscript Python decorators to intercept the taint sources, taint sinks, sanitizers and parsers at runtime. A Decorator is a Python feature, a form of meta-programming thats allows the developer to extend the functionality of existing code at runtime without permanently modifying it.

Using dynamic taint analysis, we monitor the information flow during execution time and record all sanitizers applied. A security flaw is reported immediately when tainted data reach a simple HTTP response or a raw SQL query. However, if tainted data are passed to a Django template we invoke the server-side parser. We mark these data with an annotation inside the template and we invoke the Model Browser & Sanitization Verifier modules. Model browser parses the HTTP response produced in the server-side to determine the browser context of tainted values. Sanitization Verifier validates the discovered sanitization sequences and browser context sequences.

In the effort to deploy the taint analysis library we discovered an issue in the approach of Conti & Russo. In some cases, taintedeness of data can be revoked as tainted information flows from a taint source to a taint sink. As a result, untainted data will reach the taint sink and, therefore, no flaw will be reported even though it might exist.

To tackle this problem, we introduce a new customised version of static taint analysis that builds an Abstract Syntax Tree (AST) of a target code file and, beginning from the taint source line, it parses the code line by line until it reaches the end of local scope or a return call. According to Rice's theorem, static analysis is undecidable, so we make a compromise between precision and decidability. We use approximate answers and we consider a list of assumptions.

At the end, inevitably, static taint analysis inherently tends to present some False Positives. Nevertheless, based on the test cases and senarios we have conducted, we argue that our tool successfully reports the majority of security flaws.

SUBJECT AREA: Vulnerability Detection, Hybrid Taint Analysis

KEYWORDS: security, cross-site scripting (xss), sql injection, dynamic taint analysis, static taint analysis

ΠΕΡΙΛΗΨΗ

Στην παρούσα πτυχιακή εργασία υλοποιούμε έναν επόπτη εκτέλεσης (execution monitor) που συνδυάζει Υβριδική Ανάλυση και Συντακτική Ανάλυση πλευράς Διακομιστή προκειμένου να ανακαλύψουμε ευπάθειες XSS σε template rendering, ευπάθειες XSS σε απλές αποκρίσεις HTTP και πιθανά SQL injections σε ακατέργαστα ερωτήματα SQL σε διαδικτυακές εφαρμογές Django.

Παρατηρείται ότι πολλές βιομηχανικές λύσεις εφαρμόζουν αυτο-απολύμανση χωρίς ευαισθησία περιβάλλοντος (context-insensitive auto-sanitization) ως κύρια στρατηγική άμυνας. Ωστόσο, αυτό παρέχει μια λανθασμένη αίσθηση ασφάλειας, καθώς τα μη αξιόπιστα δεδομένα πρέπει να απολυμανθούν διαφορετικά με βάση το περιβάλλον τους (browser context). Επομένως, η αναγκαιότητα ελέγχου context-sensitive ευπαθειών είναι έκδηλη. Επιπλέον, πρέπει να δοθεί μέριμνα τόσο σε απλές αποκρίσεις HTTP καθώς δεν υπάρχει αντίστοιχη πολιτική auto-sanitization όσο και σε ακατέργαστα επερωτήματα SQL (raw SQL queries) που χρησιμοποιούνται όταν το Django ORM αποδεικνύεται ανεπαρκές.

Παρέχουμε ένα εργαλείο ανάλυσης μέσω μιας βιβλιοθήκης εξ ολοκλήρου γραμμένης σε Python με βάση την προσέγγιση των Conti & Russo [2], αργότερα υιοθετηθείσα από τους Steinhauser & Tůma [1]. Δεν απαιτούνται τροποποιήσεις στον διερμηνέα. Επιστρατεύουμε Python decorators προκειμένου να ανακοπούν οι πηγές εισαγωγής ευαίσθητων δεδομένων (taint sources), οι καταβόθρες που καταλήγουν τα δεδομένα (taint sinks), οι απολυμαντές (sanitizers) και οι συντακτικοί αναλυτές (parsers) κατά την εκτέλεση της εφαρμόγης. Ένας decorator είναι μια μορφή μετα-προγραμματισμού που επιστρέπει στον προγραμματιστή να επεκτείνει τη λειτουργικότητα υπάρχοντος κώδικα κατά τον χρόνο εκτέλεσης.

Χρησιμοποιώντας δυναμική ανάλυση εποπτεύουμε τη ροή πληροφοριών κατά το χρόνο εκτέλεσης και καταγράφουμε όλους τους εφαρμοσθέντες απολυμαντές. Μια ευπάθεια αναφέρεται αμέσως αν τα μολυσμένα δεδομένα έχουν φθάσει σε μια απλή απόκριση HTTP ή σε ένα raw SQL query. Ωστόσο, αν μολυσμένα δεδομένα έχουν φτάσει σε ένα Django template καλούμε τον συνακτικό αναλυτή πλευράς διακομιστή. Αναλυτικότερα, καλούμε τις λειτουργικές μονάδες Model Browser & Sanitization Verifier, προσδιορίζουμε το browser context των μολυσμένων μεταβλητών και επαληθεύουμε τα αποτελέσματα.

Στην προσπάθεια μας ανακαλύψαμε ένα ζήτημα στην προσέγγιση των Conti & Russo. Σε ορισμένες περιπτώσεις, η προσημείωση μολυσματικότητας των δεδομένων (taintedeness) μπορεί να ανακληθεί καθώς οι μολυσμένες πληροφορίες ρέουν από μια πηγή προς μια καταβόθρα. Ως αποτέλεσμα, δεδομένα χαρακτηρισμένα ως μη μολυσμένα θα φτάσουν στο καταβόθρα και δεν θα αναφερθεί καμία ευπάθεια παρόλο που μπορεί να υπάρχει.

Για την αντιμετώπιση του προβλήματος, εισάγουμε μια νέα προσαρμοσμένη έκδοση στατικής ανάλυσης που δημιουργεί ένα Abstract Syntax Tree (AST) ενός αρχείου κώδικα και το αναλύει γραμμή γραμμή. Σύμφωνα με το θεώρημα του Rice, η στατική ανάλυση είναι μη επιλύσιμη οπότε συμβιβαζόμαστε μεταξύ ακρίβειας και μη επιλυσιμότητας και χρησιμοποιούμε επιλύσιμες κατά προσέγγιση απαντήσεις.

Τελικώς, αναπόφευκτα η στατική ανάλυση τείνει εγγενώς να παρουσιάζει κάποια ψευδώς θετικά αποτελέσματα. Εντούτοις, βασιζόμενοι στα τεστ και τα σενάριο που διεξήγαμε, υποστηρίζουμε ότι το εργαλείο μας αναφέρει επιτυχώς την πλειόνοτητα των ευπαθειών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανίχνευση Ευπαθειών, Υβριδική Ανάλυση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ασφάλεια, cross-site scripting (xss), sql injection, δυναμική ανάλυση, στατική ανάλυση

To my father
who supported and encouraged me
throughout my time at university.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Alex Delis for his guidance and assistance in the preparation of this thesis.

CONTENTS

FIGURES LIST

TABLES LIST

1. INTRODUCTION

Nowadays, technological advancements and the development of cloud services have allowed a significant increase of the activities performed online. Users capabilities have been skyrocketed from popular but common services e.g. watching videos, listening to music, socializing, calling for transportation etc. to many security sensitive services e.g. e-banking & finance, e-commerce, e-government etc. whose compromise could lead to devastating results.

As a matter of fact, several web attacks have been launched the previous years with a considerable economic and social impact for the companies and the users. It is calculated that more than 14,717,618,286 data records have been lost or stolen since 2013 and it is estimated that 6,500,715 data records are compromised every day meaning that 75 records are compromised every second [16]. As a result, we realize that web attacks is not just a mere scientific issue studied only by field experts and technology enthusiasts, but a tool that can be powerfully used for financial purposes, hactivism, espionage, cyber-terrorism or cyberwarfare among sovereign states.

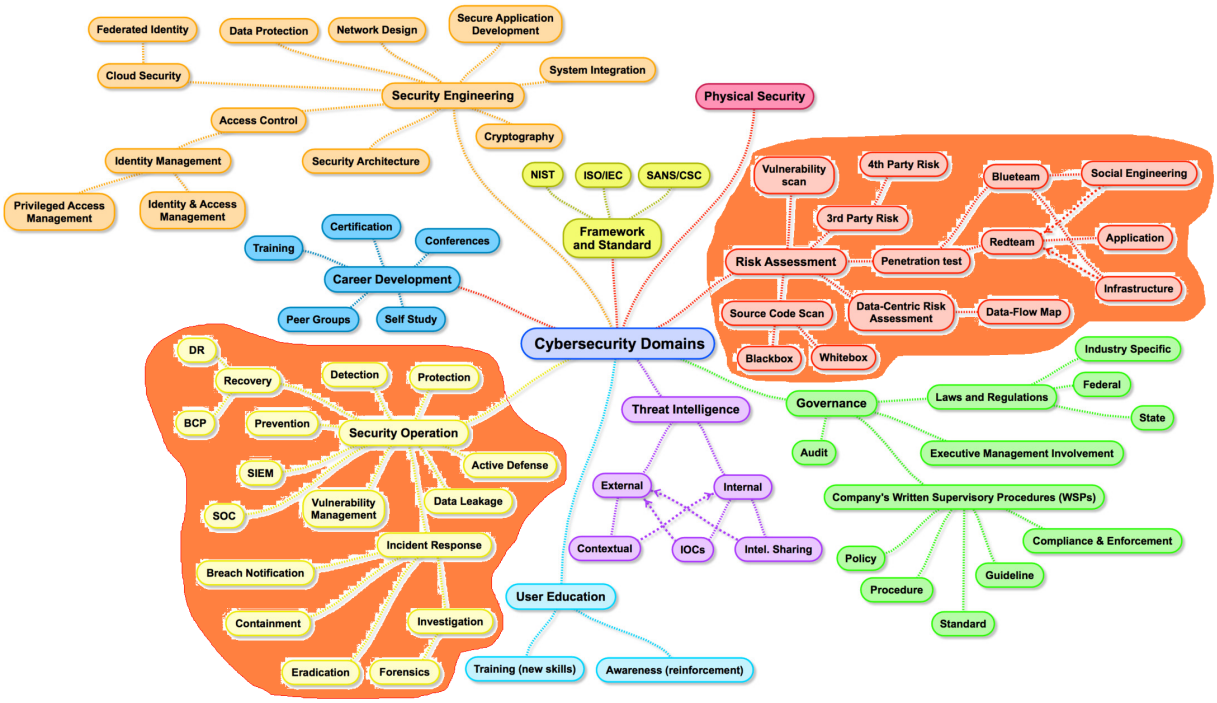

Figure 1: The Map of Cybersecurity Domains (version 2.0) [17]

In this thesis we focus on a specific category of web attacks, code injection. Specifically, we provide a tool for detection of security flaws through dynamic taint analysis and server-side parsing (Security Operation in figure 1) combined with static taint analysis (Source Code Scan in figure 1). We aim to build an execution monitor that discovers SQL injection flaws as well as Cross-site Scripting (XSS) flaws in respect to context-sensivity. As [10] points out, these attacks are placed at the top in terms of most critical web-app security risks.

We organize this thesis in 7 sections. First, we present the theoretical framework of XSS and SQL injection in section 2. Subsequently, we describe Python Decorators and Dynamic Taint Analysis (taint tracker) in section 4, Model Browser and Sanitization Verifier in section 5 and Static Taint Analysis in section 6. In section 7 we present experimental results and test cases. Last, in section 8 we sum up what we have covered in this thesis.

2. THEORETICAL FRAMEWORK

2.1 Cross-site Scripting (XSS)

2.1.1 Introduction to XSS

Cross-site Scripting (XSS) is a type of security vulnerability found in web applications. XSS attack allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites. XSS is listed 3rd in *The Ten Most Critical Web Application Security Risks* report by OWASP in 2013, while dropped 7th in 2017 [10]. It is the 2nd most prevalent issue in the OWASP Top 10, and is found in around two-thirds of all applications.

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript.

Typical XSS attacks include session stealing, account takeover, MFA bypass, DOM node replacement or defacement (such as trojan login panels), attacks against the user's browser such as malicious software downloads, key logging, and other client-side attacks [30].

2.1.2 Forms of XSS

There are three forms of XSS:

1. **Reflected or Non-Persistent XSS**: Occurs when user input is immediately returned by a web application in a response that includes some or all of the user-provided data. The data have neither been made safe to render in the browser, nor permanently stored [10].
2. **Stored or Persistent XSS**: The application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk [10].
3. **DOM Based XSS**: The attack payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner [31]. We are **not** addressing this issue in this thesis.

2.1.3 XSS Context-Sensivity

XSS attack has been studied by many authors and industry has worked towards a solution, so a fair question arises why it is worth dealing with it. As a matter of fact, auto-sanitization techniques are enforced to guarantee the defense success [12].

However, it is observed that many industry solutions implement context-insensitive auto-sanitization which, on the one hand, shifts the burden of ensuring safety against XSS from developers, on the other hand, provides a false sense of security [1, 3]. Untrusted data need to be sanitized differently based on their browser context in HTML document. For example, the sanitization requirements of Javascript code are different from those of an HTML tag. As a result, a sanitizer that may be safe for use in one context may be unsafe for use in another.

Therefore, to achieve security, auto-sanitization must be context-sensitive. Lack of auto-sanitization obliges us to rely on developers to pick a sanitizer consistent with the context. This policy is error-prone and could subvert the entire application's integrity. For that reason, a detection or avoidance mechanism should be installed to monitor the program's behavior.

2.2 SQL Injection

SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQLi Server will execute all syntactically valid queries that it receives. Even parameterized data can be manipulated by a skilled and determined attacker [9].

Code injection, as an general attack, is listed 1st in *The Ten Most Critical Web Application Security Risks* report by OWASP in 2017, remaining in the top of the list since at least 2010 [11]. For that reason, SQL injection poses an attractive candidate for study.

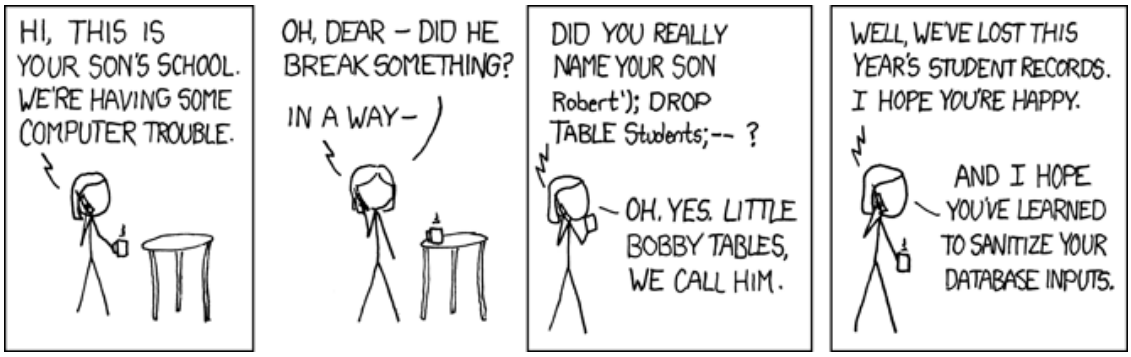

Figure 2: SQL Injection by Little Bobby Tables (https://xkcd.com/327/)

SQL injection remains a fairly well acknowledged attack. Particularly, in October 2015, personal details of 156,959 customers from British telecommunications company TalkTalk were leaked and company got a record £400,000 fine for failing to prevent the attack [13]. In July 2012 a hacker group was reported to have stolen 450,000 login credentials from Yahoo! by using a union-based SQL injection technique [15]. In 2009, US prosecutors charged a man with stealing data relating to 130 million credit and debit cards using an SQL injection attack [14]. So, code injection has a potentially significant social and economical impact.

2.3 Django: The web framework for perfectionists with deadlines

2.3.1 Introduction to Django

Django is a Python-based open-source web framework based on the Model-View-Template (MVT) architecture. Django adopts minimalistic design philosophies such as loose coupling, less code, quick development and extensibility aiming to become a web framework of the 21st century. It has been used by large companies such as Instagram, Disqus and Mozilla [35].

Apart from the ease of use and popularity that Django presents, it is an attracitve candidate for our study. Not only does Python facilitate Dynamic Taint Analysis but also Django supports a template language with context-insensitive auto-sanitization and manual sanitization, where we can demonstrate better the context-sensitive XSS flaws.

For the needs of the thesis, we use Python3.9 and Django version 3.1.2 in Ubuntu Linux, which are the latest versions at the time of production (Summer-Autumn 2020). We are aware that technologies are evolving fast so it is a matter of time until a new version is published. Nevertheless, we believe that our tool can perform adequately in the new versions, with minor or no changes at the code, provided that no radical changes are introduced in Python or Django.

2.3.2 XSS in Django

As discussed before, to achieve security, auto-sanitization must be context-sensitive. Since Django enforces auto-sanitization on its template language, the **1st** major part of our security flaw analysis deals with context-sensitive XSS flaws in Django Template Language (DTL).

DTL is a feature for keeping the application logic separate from design. Application code is developed and maintained separately from the presentantion code displayed by the client browser (HTML/XML, JavaScript, JSON etc). Thus, code is easily developed, maintained and refactored. Template System is responsible for rendering python data into server responses. For example, HTML document can have the format of the figure 3. Python inserts all the relative data such as *description, blog_entries* etc. and serve it to the client.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <link rel="stylesheet" href="style.css">
5      <title>{% block title %}My amazing blog{% endblock %}</title>
6  </head>
7
8  <body>
9      <h1>{{ description|upper }}</h1>
10     {% block content %}
11     {% for entry in blog_entries %}
12         <h2>{{ entry.title }}</h2>
13         <p>{{ entry.body }}</p>
14     {% endfor %}
15     {% endblock %}
16 </body>
17 </html>
```

Figure 3: Example of Django Template Language code snippet for HTML document

Django documentation claims that Django templates provide sufficient measures against the majority of XSS attacks [32]. Nevertheless, it is important to understand the nature of this protection and its limitations. Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. In fact, Weinberger et al. [3] suggest that Django's auto-sanitization mechanism fails to correctly protect between 14.8% and 33.6% of an application's output sinks.

```
1  <p id="unique"></p>
2  <script>
3      var par, query;
4      par = document.getElementById ("unique");
5      query = "{{ name }}";
6      par.innerHTML = query;
7  </script>
```

Figure 4: Example of XSS attack in Django template [1]

In figure 4, if *name* is, for example, *<script>alert(1)</script>* then auto-sanitization will work. On the contrary, if user enters *\x3cimg src=\x22N/A\x22 onerror=\x22 alert(1) \x22 /\x3e* which translates to **, it will tremendously fail. It is required a *{{ name | escapejs }}* for the sanitization to work. This example makes the case for control over context-sensitive flaws, strikingly evident.

On the contrary, for the **2nd** major part of our security flaw analysis, Django has no auto-sanitization policy regarding simple HTTP responses like *HttpResponse()*. This makes it even difficult to eliminate the possibilities of an XSS attack, especially if a developer decides to use a simpe HTTP response instead of template rendering. Consequently, it is indisputable that attention has to be given also to HTTP responses in which we search for context-insensitive XSS flaws.

2.3.3 SQL Injection in Django

Django documentation claims that Django's Object-Relational Mapping (ORM) are protected from SQL injection since their queries are constructed using query parameterization. A query's SQL code is defined separately from the query's parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver [32]. So Django ORM is reported to be highly resistant to SQL injection.

However, in this thesis we are concerned more about the ways a developer can bypass the Django ORM rather than protecting the ORM itself. Raw SQL queries are often used when ORM is proved not enough. Django provides two ways of performing raw SQL queries [33], which might pose a security risk if not handled correctly:

1. the *Manager.raw()* to perform **raw queries** and return model instances:

```
1  >>> Person.objects.raw('SELECT * FROM newsletter_name where  name = ' +
       param)
```

Figure 5: Example of raw query in Django

In figure 5, *Person* corresponds to a model table name i.e. a model class which is mapped to a database entity. Supposing that *param* variable is user-defined, we have a possible SQL injection attack.

2. the *django.db.connection* to **execute custom SQL** directly:

```
1  with connection.cursor() as cursor:
2      cursor.execute("SELECT * FROM newsletter_name WHERE name = " + param)
3      row = cursor.fetchone()
```

Figure 6: Example of custom SQL execution in Django

In figure 6 we perform a custom SQL query using the *param* variable which might be user-defined and, therefore, subject to SQL injection.

Our purpose is to perform code analysis and figure out if there is a possibility of an SQL injection attack. This is the **3d** major part of our security flaw analysis.

3. TOOL ARCHITECTURE

3.1 Existing work & What's new

Our analyser tool consists of 5 components. Each one corresponds to a Python module. The initial design idea came up from the approach of Steinhauser & Tůma with 3 main modules (Dynamic taint analysis & Python Decorators, Model browser, Sanitization verifier).

What's new:

- We refactor the current design structure and individualize Python Decorators. This approach helps us make the analyser customisable to changes and provides cleaner and easily maintainable code.
- We improve the tracking and reporting capabilities of the existing Dynamic Taint Analysis module. The initial work is primarily based on discovery of context-sensitive XSS flaws in template languages. In our implementation, there are 2 new benefits. Firstly, we extend the scope of XSS flaw analysis to simple HTTP responses. As a result, we are able to monitor all the possible ways Django produces a server response. Secondly, we take into consideration a new type of web attack, apart from XSS, the SQL injection. We focus on tracking potential SQL injection in raw SQL queries.
- We introduce a new component, Static Taint Analysis. This complementary module discovers flaws where Dynamic Taint Analysis is unable to do so.

Decorators can be modified any time to limit or extend the analyser's tracking scope. Consequently, if additional framework or programming capabilities arise, then it is a matter of settings change for our tool to include new changes and work. Last, the tool runs simultaneously with Django server.

3.2 Component structure

We sum up the operational behavior of our own design proposal below and we perform deeper analysis in the next sections of this thesis:

1. **Python Decorators**

   This module injects instrumentation code to the subject application in order to intercept the taint sources, taint sinks, sanitizers and parsers at runtime. A Decorator is a Python feature, a form of meta-programming thats allows the developer to extend the functionality of existing code at runtime. So they help us monitor the behavior of a Django application without permanently modifying its source code.

2. **Dynamic Taint Analysis (Taint Tracker)**

   This module monitors the information flow during Django execution time and records all sanitizers applied. Taint tracking reports a flaw immediately when tainted data have reached a simple HTTP response or a raw SQL query. However, the report is delayed if tainted data are passed to a template. In that case, tainted data are marked with an annotation inside the template (Global Registry) and the Model Browser & Sanitization Verifier modules are invoked. If Taint Analysis fails to discover a flaw, then taintedeness of data might has been be revoked by Python during execution flow, so we invoke the Static Analysis module.

3. **Model Browser**

   This module parses the HTTP response produced in the server-side to determine the browser context for potentially tainted values, that have reached a taint sink, and the position of those is marked with a unique annotation (Global Registry).

4. **Sanitization Verifier**

   This module validates the discovered sanitization sequences as well as browser context sequences and prints the final report.

5. **Static Taint Analysis**

   This module is a customised version of static taint analysis with a goal to search for XSS and SQLi flaws when dynamic taint analysis is unable to do so. It comes as a handy solution to an issue discovered by us regarding the approach by Steinhauser & Tůma based on the prototype of Conti & Russo. In their approach, taintedeness of data can be revoked, during dynamic taint analysis, as tainted information flows from a taint source to a taint sink. As a result untainted data will reach the taint sink and, therefore, no flaw will be reported. To deal this situation, we enforce static analysis that does not exist in the existing approaches.

3.3 Activity diagram

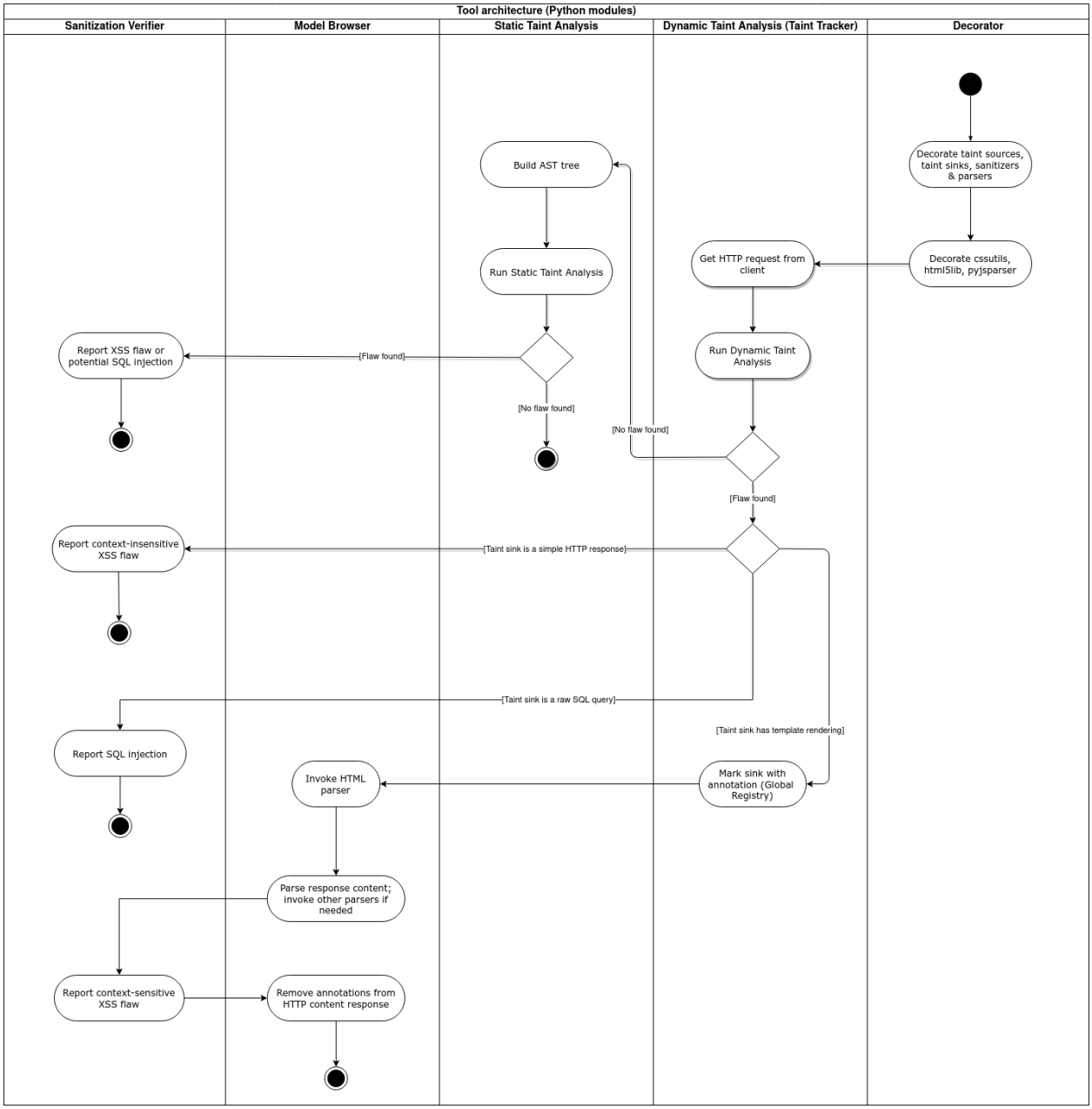

Figure 7: UML activity diagram of tool architecture for our implementation proposal

Figure 7 shows a UML activity diagram which describes the activity sequence of our analysis tool and the transactions among components (diagram pools). Specifically, analysis starts by setting up Python decorators along with Django server initialization. After, when Django receives an HTTP request from the client, our tool performs dynamic taint analysis to track down potentially tainted data, in parallel to Django request processing. If a flaw is found, then we either report the flaw or run server-side parsing, depending on the type of taint sink. For simple HTTP response or raw SQL query as taint sink, we report a flaw immediately. For template as taint sink, we run parsing (model browser, sanitization verifier). If no flaw found, we run static analysis. Tool stops when Django is terminated or analysis is completed having found a flaw or not.

4. DYNAMIC TAINT ANALYSIS

4.1 Python Decorators

A Decorator is any callable Python object than can modify or extend a function, class or method at runtime. It is a form of meta-programming thats allows the developer to wrap an aforementioned entity in order to extend its behavior without permanently modifying it.

Decorators are extremely useful for dynamic taint analysis since they can inject instrumentation code to the subject application and intercept the taint sources, taint sinks, sanitizers and parsers at runtime. They allow our tool to obtain runtime access at objects and structures of Django. Also, the facilitate the discovery of the lines inside the developer's code where malicious information appears, the lines where it ends up and how this information is processed. Hence, it is a way to motinor the information flow and check for security flaws. A general example is the following:

```
1  def decorate_function(func):
2      def wrapper(*args, **kwargs):
3
4          # wrapped function e.g. taint sink
5          func(*args, **kwargs)
6
7          # new functionality
8          print("This is the decorator")
9
10     return wrapper
11
12 def target_function():
13     # ..code here
14
15 target_function = decorate_function(target_function)
```

Figure 8: Simple example of Python decorators

which is equivalent to:

```
1  @decorate_function
2  def target_function():
3      # ..code here
```

Figure 9: Simple example of Python decorators with annotation

Based on the example of figure 8, *target_function* (line 12) is a function of our interest (taint source, taint sink, sanitizer or parser) and *decorate_function* is the decorator (line 1). Decorator runs the original function *func* and introduces new functionality (in the example it is just a print function). In line 15, we reassign the returned object from the decorator, i.e. the *wrapper* function, to *target_function*. In that way, every time we invoke *target_function*, the wrapper function will be executed instead.

4.2 Dynamic Taint Analysis

4.2.1 Introduction to Dynamic Taint Analysis

Dynamic taint analysis is a technique used to monitor the information flow during execution time and report security flaws.

Taint analysis begins from the code locations where potentially malicious inputs appear. These locations are called *taint sources*. All values produced by taint sources are considered tainted. For example, when a tainted string is concatenated with another string, the final string is marked as tainted. Tainted values are tracked along data-flow paths until they reach security sensitive entities, called *taint sinks*. During data flow, tainted data might pass through a *sanitizer*, which removes the taintedeness from the value making it harmless. Last, when we are about to return the HTTP response to client, *parser* processes the model browser of the response content in case of template rendering.

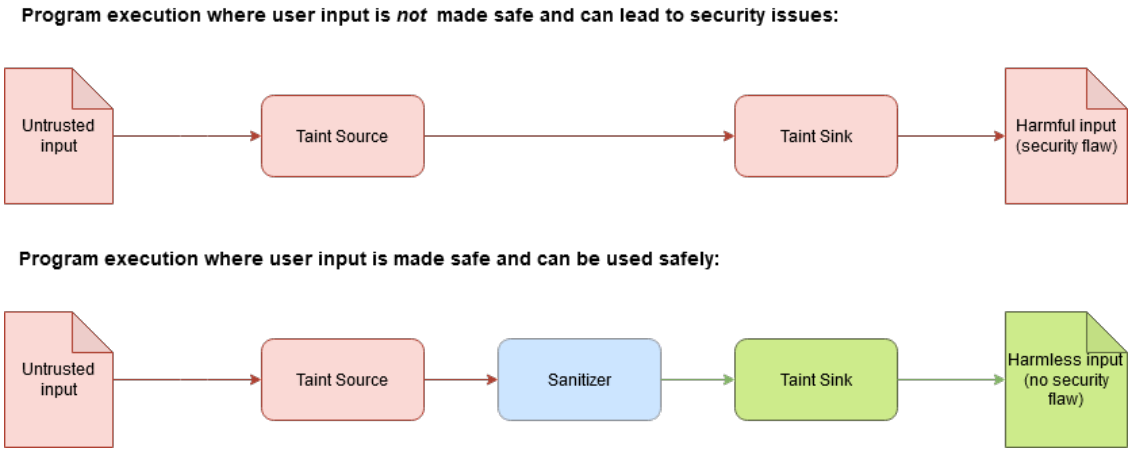

Figure 10: Taint analysis visualization

The purpose of dynamic taint analysis is to record the sequence of applied sanitizers as information flows from a taint source to a taint sink.

4.2.2 Preparatory work

Our dynamic taint analysis implementation is based on the approach presented by Conti & Russo [2], in which the authors provide a novel and elegant solution on dynamic taint analysis for Python via a library entirely written in Python. The work is, later, adopted by Steinhauser & Tůma [1] and, subsequently, by us. The major advantages of this approach is the flexibility and portability, as the analysis runs onto the server and there is no need to intervene on Python interpreter. However, as we see in section 6, there are some cases where the approach fails and the need to take additional measures arises.

Discussed before, we extend the vulnerability scope of the existing approaches to include 2 new targets: 1) context-insensitive XSS flaws in simple HTTP responses and 2) SQL injection in raw SQL queries.

We track both string (which are the obvious holder types for an attack payload) and numerical data types. Additionally, container data structures such as lists, tuples, dictionaries, and their descendants maintain the taint information in the contained elements. When a function propagating taint data returns a container, the taint information is stored recursively in all elements of the container [1]. Last, we ignore implicit flows which are harmless [8] as well as boolean data types.

4.2.3 Implementation internals

For dynamic taint analysis to work, we have to figure out a way to distinguish tainted data from clean data. The key idea is to create special classes that can be recognized by Taint Tracker and deal with taint information. In general, these classes are extact copies of the original classes of the data, but they have enhanced functionality.

As a result, we need to focus on two major core parts: firstly how to generate taint-aware classes based on built-in classes, and, secondly how to generate taint-aware class methods that propagate taint information.

We begin with the first core part and how to keep track of taint information for built-in classes. The module defines subclasses of built-in classes in order to indicate if values are tainted or not. An object of these subclasses posses an attribute to indicate the sanitizer sequences associated to it. These subclasses are taint-aware and carry the name *Taint AwareClass*. It is important to note that there is not a single *TaintAwareClass*. Each taint-aware class we build based on a built-in class carries the name *TaintAwareClass*.

The second core part handles the redefintion of taint-aware class methods. Specifically, the methods inherited from the built-in classes are redefined in order to propagate taint information. So, when called by the dynamic dispatch mechanism they return taint-aware objects. In the sanitizer sequence field, we include the union of sanitizer sequences found in the arguments and the object's calling the method [2]. It is important to note that depending on the method semantics, some methods are redefined, while others not. For example, the extended classes do not override taint-free methods such as *__cmp__* or *__eq__* and preserve the original implementation instead. Similar taint-propagating wrappers are applied to global functions such as *ord* or *chr* to cover all places where the taintedness can be transmitted [1].

Having defined the taint-aware classes, now, we have to figure out a way to monitor information flow as Django application code is executed. Potentially tainted values that appear in HTTP request fields are provided by the user. For example, in figure 11, *firstname* or *lastname* fields of POST request might contain malicious code and the developer neglects to sanitize them. As a result, an XSS attack or SQL injection might occur by simply inserting attack payload in the GET reuqest below.

So, dynamic taint analysis begins the moment we retrieve these data (lines 13-16 in figure 11). These are the taint sources. These variables are not of type string or unicode, but *TaintAwareClass* so we can recognize potentially malicious data. Application code runs seamlessly. When we reach a taint sink (line 21), we check if data are still of type *TaintAwareClass*. If so, we have found an security flaw. If not, then some form of sanitizer has been applied in the process (line 18).

```
1  from django.http import HttpResponse
2  from django.shortcuts import render
3
4  class SignupForm(forms.Form):
5     username = forms.CharField(label='Enter usernname', max_length=20)
6     firstname = forms.CharField(label='Enter first name', max_length=20)
7     lastname = forms.CharField(label='Enter last name', max_length=20)
8     password = forms.CharField(widget=forms.PasswordInput())
9
10 def signup(request):
11    if request.method == 'POST':
12       # taint sources
13       username = request.POST.get("username")
14       password = request.POST.get("password")
15       firstname = request.POST.get("firstname")
16       lastname = request.POST.get("lastname")
17
18       # ..user data process
19
20       # taint sink
21       return HttpResponse("<html><body><p> Welcome, " + firstname + " " +
          lastname + "</p></body></html>")
22    elif request.method == 'GET':
23       signupform = SignupForm()
24       return render(request, 'signup.html', {'signupform': signupform})
25    else:
26       return HttpResponse("Method not supported")
```

Figure 11: Example of Django view code that signups a new user

With the help of decorators we intercept all the functions we discussed about, above. Each decorator function carries out a different set of operations. First, the decorator executes the original function and then:

1. **Taint Source**: returns a taint-aware object of class *TaintAwareClass*, based on the original result of the taint source, in order to propagate taint information.
2. **Sanitizer**: transforms the result object of the sanitizer function into a *TaintAwareClass* object affiliating it with the browser context the sanitizer is related to, and the existing sanitizer sequences. That is because objects passed in the sanitizer function might be taint-aware but the result object of the sanitizer might not, so taintedeness may be lost and needs to be restored.
3. **Taint Sink**:

   (i) if the result returned by the original function is taint-aware (*TaintAwareClass*) we are dealing with some form of **template rendering** (*render(), Response(), TemplateResponse() etc*) and, subsequently, we mark the sink location with an unique annotation inside the template and store it to Global Registry (which is a Python dictionary). Process continues with the Model Browser & Sanitization Verifier modules where annotations are discovered and affiliated with a browser context sequence. Finally, a context-sensitive XSS flaw is reported.

(ii) if the taint sink receives taint-aware objects as arguments, then we are dealing with:

- a **simple HTTP response** (*HttpResponse(), StreamingHttpResponse()..*). Context-insensitive XSS flaw is reported immediately.
- a **raw SQL query** (*Manager.raw(), Manager.extra(), cursor.execute() etc*). Potential SQL injection is reported immediately.

4. **Parser**: invokes the main function of the Model Browser module in case of template (case 3.i).

We realize that the elements from 1 to 4 work as a chain that controls the information flow during program execution. In other words, it is the implementation of taint analysis visualization presented in figure 10. The *parser* element is the link between taint analysis and side-server parsing, which is presented in the next section.

5. SERVER-SIDE PARSING

5.1 Introduction to Server-side Parsing

Server-side parsing concerns the discovery of security flaws only in Django templates and it has 2 purposes. The first is to determine the browser context for all potentially tainted values that have reached a taint sink. The second is to check whether the applied sanitizations are sufficient for the particular browser context. For that reason, we examine the HTTP response content before it is returned to the client using a set of customised web scraping frameworks. Server-side parsing is used to discover context-sensitive XSS flaws in template responses.

5.2 Model Browser

5.2.1 Model Browser architecture

In the effort to facilitate the examination of the HTTP response content, we conscript existing Python utilities. Django templates are basically written in HTML, CSS and JavaScript in an *.html* file. Therefore, we consider 4 different browser contexts: HTML, JavaScript, CSS and URL and we construct 4 independent parsers. Except for the URI parser, each parser builds a Document Object Model (DOM) and traverses it. So, on the one hand, for building the HTML DOM we use html5lib [19] running over Beautiful Soup 4 [23]. For the CSS DOM we use cssutils [20] and for the JavaScript DOM we use PyJsParser [22]. The URL parser, on the other hand, is a manually written lexer which analyzes only the *javascript:* and *data:* URLs.

The main goal of server-side parsing is to determine the browser context in which the tainted values appear. As a result, we recursively or iteratively parse the content of each HTTP response and analyse the browser code based on syntax of the aforementioned web technologies.

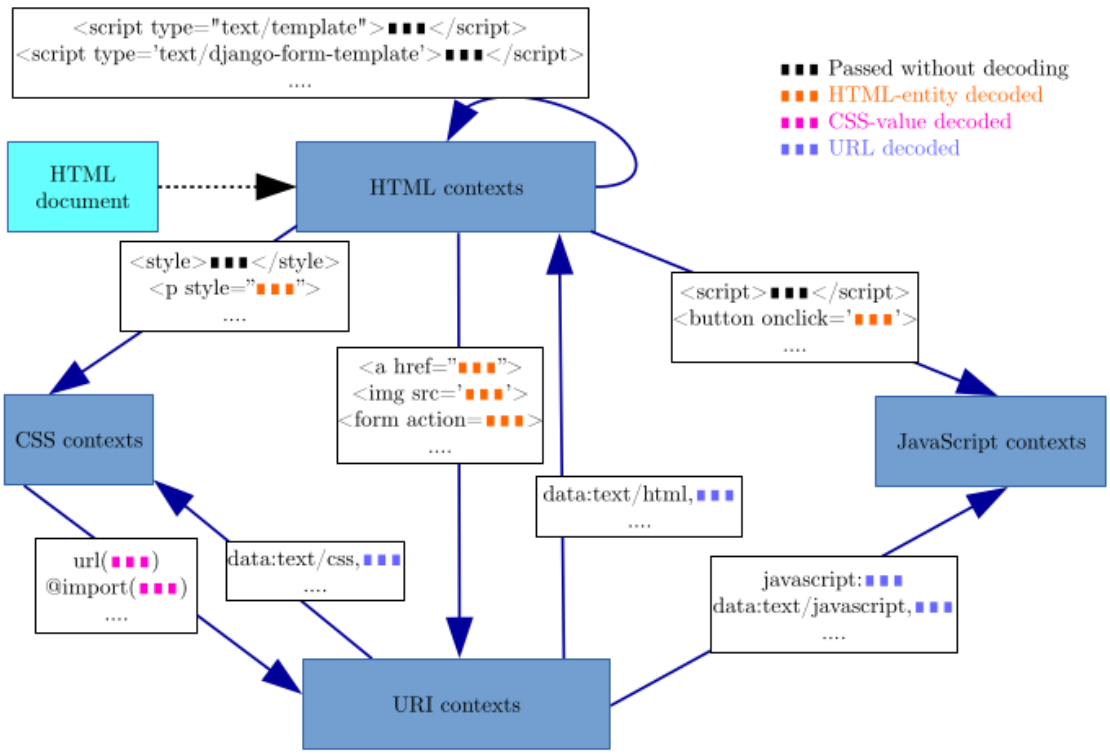

Figure 12: Transitions between parsers in the model browser [1]

The process starts with the HTML parser, which receives the HTTP response content and invokes any of the other 3 parsers, if needed. For example, when the HTML parser reaches a url invokes the URL parser. The latter discovers a *javascript:* keyword and, respectively, invokes the JavaScript parser. Transitions among parsers in the model browser are depicted in figure 12. Each nested parser invocation is mapped at a maintained browser context sequnece of a dicitonary type, called Detected.

For example, we assume we have a simple GET request from client, presented in figure 13. In figure 14, we have a simple template response that Django will send back. The request contains a parameter *link* which can be freely modified by the user. Django will put the link's content in line 5 of the response (figure 14) and send it to client.

It is evident that this code is easily exploitable as link can be anything, even executable JavaScript code: *?link=javascript:alert('Hi');*. Our analysis tool has tracked that malicious data have reached the template in the previous step and it has marked the data with an annotation. So, in this module we compute the browser context sequence of *link*. Specifically, we come up with a browser context sequence as [HTML_JS_DATA (code e),JS_CODE (code J)], meaning that data have passed from HTML to JavaScript environment and they rely now on the latter.

```
1  GET /hello/?link=javascript:alert(%27Hi%27); HTTP/1.1
2  Host: 127.0.0.1:8000
3  User-Agent: Mozilla/5.0 (Linux) Gecko/20100101 Firefox/84.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5  Accept-Language: el-GR,el;q=0.8,en-US;q=0.5,en;q=0.3
6  Accept-Encoding: gzip, deflate
7  DNT: 1
8  Connection: keep-alive
9  Cookie: csrftoken=
       IVOLsIoIT89Xg5soR1nWziFdHm4JS7FFXvfdUS4xigpvy91hBEvzHmoCIFUUbEKp

10 Upgrade-Insecure-Requests: 1
```

Figure 13: Simple GET request sent by client-side

```
1  <!DOCTYPE html>
2  <html>
3     <body>
4        <h1> description </h1>
5        <a href=" {{ link }} ">Click Me!</a>
6     </body>
7  </html>
```

Figure 14: Example of XSS attack payload hidden in data: URI scheme

The main goal of our parsers is to search for annotations from the Global Registry and update the current browser context sequence. An annotation is included in the Global Registry when it is written to the HTML output. At the end of parsing, all annotations should therefore be found and associated with both the relevant taint information and the relevant

browser context sequence. This information is then processed by the sanitization verifier [1]. At the end, annotations are removed and the initial response content is restored.

5.2.2 HTML Parser Implementation

The HTML parser transforms a complex HTML document into a complex tree of Python objects with the help of Beautiful Soup. We selected html5lib because it is extremely flexible, it parses pages the same way a web browser does and creates valid HTML5 [24].

We deal with 5 types of objects so we include everything we encounter in HTML document [24]:

1. **BeautifulSoup**: It represents the parsed document as a whole and stores the parse tree we traverse
2. **Tag**: According to W3C Recommendation on HTML 5.1 (2nd Edition), Tags are used to delimit the start and end of elements in the markup. The presence of a script or style tag demands the direct invocation of the JavaScript or CSS parser, respectively. Else, we simply need to process the tag attributes and the tag contents separately (figure 15).

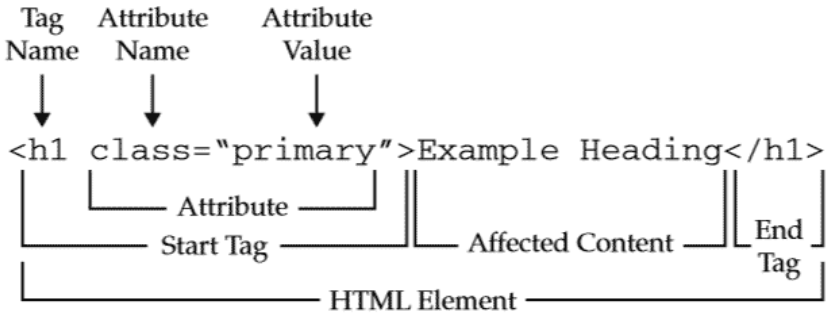

Figure 15: HTMl tag example from https://tutorial.techaltum.com/htmlTags.html

A tag attribute might be related to URL (e.g. *href, src*), JavaScript (e.g. *onclick, onerror*) or CSS (e.g. *style*) content and if so, we invoke the respective parser and feed it with the attribute's content.

Tag attribute values may be wrapped by double (”), single (’) or no quotation marks. In order to maintain an updated and precise browser context sequence we need to patch those html5lib parsing states related to string quotation, so we can recognize the quoting type of the html tag attributes.

3. **NavigableString**: A string corresponds to text within a tag. Handled as string
4. **Comment**: Subclass of NavigableString which corresponds to HTML comment and handled as string

Also, Beautiful Soup defines classes for anything else that might show up in an XML document e.g. CData, ProcessingInstruction, Declaration, and Doctype, from which we keep **Doctype** and is handled as string.

5.2.3 JavaScript Parser Implementation

The implementation of the JavaScript parser seems rather simple compared with the rest of parsers. For discovering annotations, we build a parse tree in the form of Python dictionary and through recursive parsing we end up in processing literals. Literals represent

values in JavaScript. These are fixed values that we actually provide in a script [26]. Consequently, we only have to examine the literal for annotations from the Global Registry.

One concern we need to express in this point, is that a JavaScript string can nest any other context. As Steinhauser & Tůma [1] note, detection of nested context in JavaScript strings is, unfortunately, an algorithmically undecidable problem. JavaScript is a Turing-complete language that can perform arbitrary operations with the tainted string comparing its definition with its usage. The same string might even be cloned and end up in different context combinations. Despite the approach seems incomplete, the browser context sequence is successfully kept up to date.

5.2.4 CSS Parser Implementation

The CSS parser interprets a CSS stylesheet as a set of CSS rules with the help of cssutils [21]. Most rules are known as at-rules. Briefly, an at-rule is a statement that provides CSS with instructions to perform or how to behave. Each statement begins with an @ followed directly by one of several available keywords that acts as the identifier for what CSS should do [28]. Below are listed the rules we handle, in total:

Table 1: List of CSS rules handled by CSS parser [28]

Rule	Example
STYLE	body {background-color: blue}
PAGE	@page { margin: 1cm;}
CHARSET	@charset ”UTF-8”;
IMPORT	@import url(”mystyle.css”);
MEDIA	@media only screen and (max-width: 600px) { body {background-color: blue;} }
FONT-FACE	@font-face { font-family: myFirstFont; src: url(sansation_light.woff);}
MARGIN	@top-left { content: ”123”; }
NAMESPACE	@namespace url(http://www.w3.org/1999/xhtml);
VARIABLES	@variables { BG: #fff }
COMMENT	/* a comment */

Specifically, if parser encounters an IMPORT or NAMESPACE rule, then determines the quoting state and calls the URL parser. Additionally, if encounters a STYLE, PAGE, FONT FACE or MARGIN rule it breaks it down to a declaration block and a associated group of selectors (figure 16). The selector specifies which element or elements in the HTML page the CSS rule applies to. Whereas, the declarations within the block determine how the elements are formatted on a webpage [29]. In the end, it searches for annotations from the Global Registry.

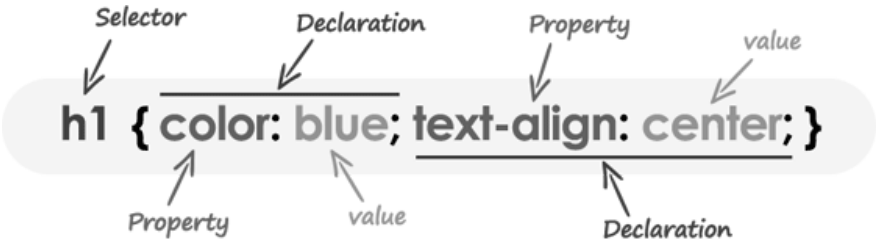

Figure 16: CSS selector & declaration syntax [29]

5.2.5 URL Parser Implementation

Web content might contain embedded URLs and if not handled correctly, it might lead to an unexpected XSS attack. If we reach a URL we invoke the URL parser. As mentioned above, the URL parser does not build any DOM. Instead, it analyzes the URL and it continues the processing only if encounters *javascript:* or *data:* keywords. This is because these keywords might contain attack payload. Other URLs are considered harmless.

To deal this situation, first of all, if we encounter the *javascript:* keyword, it suffices to invoke the JavaScript parser. Figure 17 demonstrates how we can hide malicious payload inside an HTML tag. From user-side it is just a link, but if we click it, then JavaScript code will be executed.

```
1  <a href="javascript:alert('Hi');">Click Me!</a>
```

Figure 17: Example of XSS attack payload hidden in javascript: URI scheme

Secondly, according to IETF RFC 2397 [27], data URI scheme has the following syntax: *data:[<mediatype>][;base64],<data>*. Some applications have the ability to embed media type data directly inline. Thus, an attacker can place malicious payload in the *<data>* section.

Things can get more complex if we consider that data can be encoded. For example, in line 1 of figure 18 we hide malicious payload inside an HTML tag, as in the previous example. However, in line 3 there is an equivalent representation of line 1, in which data are put in base64 encoding (*PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==* in base64 is *<script>alert(1)</script>* in UTF-8). Consequently, we realize that we just can not invoke directly any other of the 3 parsers, as previously.

```
1  <a href='data:text/html,<script>alert(1)</script>'>my link</a>
2
3  <a href='data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==’>
       my link</a>
```

Figure 18: Example of XSS attack payload hidden in data: URI scheme

For that reason, we tokenize the content based on the IETF RFC 2397 data URI scheme. We separate actual data from media type and from encoding, if exists. After, we convert encoded data to plaintext data, if necessary. Last, based on the media type we invoke the respective parser passing the dedoced data (HTML parser for *text/html*, CSS parser for *text/css* and JavaScript parser for *text/javascript*, *application/javascript* etc).

5.3 Sanitization Verifier

This module validates the discovered sanitization sequences and browser context sequences and prints the final flaw report. To do that, it validates each sequence against a set of rules.

Specifically, it checks each sequence element one by one. The first element of the sequence has to be HTML, since this is the main HTTP response language, while JavaScript and CSS are embedded in HTML. Additionally, the intermediate elements have to be successors. These are nodes who signal browser content change. For example, HTML_JS_DATA is a successor node that signals the transition from HTML to JavaScript

environment. Finally, the last element has to be a leaf, not a successor. It is important that all the sequence nodes placed right after a successor node have the browser context the successor describes. For example, after a successor node HTML_JS_DATA (moving from HTML to JavaScript) there is a node JS_CODE (JavaScript code).

6. STATIC TAINT ANALYSIS

6.1 Taintedeness revocation during Dynamic Taint Analysis

As mentioned in section 4.2, dynamic taint analysis implementation is based on the approach presented by Conti & Russo [2], later adopted by Steinhauser & Tůma [1], in which the authors provide solution on dynamic taint analysis via a library entirely written in Python.

However, in the effort to deploy the taint analysis library we discovered that, in some cases, taintedeness of data can be revoked as tainted information flows from a taint source to a taint sink. As a result, untainted data will reach the taint sink and, therefore, no flaw will be reported even though it might exist.

We are able to identify several cases of taintedeness revocation that concern mainly string data types (f-strings, str.format, str.replace), without excluding numerical data types. Having downloaded the source code of Steinhauser & Tůma and of Conti & Russo, we conducted excessive testing to study how tainted data react with non-tainted data. We concluded this delicate issue occurs when taint-aware information is mixed with non taint-aware information and Python calls the methods of built-in or custom class objects instead of the redefined methods of the taint-aware object.

For example, in figure 19 we have tainted data in variable *name* and we mix them with the clean string *Welcome user,* . In this case, Python invokes a redefined method of variable *name* that propagates taint information. As a result, variable *header* is taint-aware. Thus, the existing approach can recognize this taint information when it reaches a taint sink and report a security issue.

On the contrary, in figure 20 we have tainted data in variable *name* and we mix them with the clean string *Welcome user,* . In this case, Python does not invoke a redefined method of variable *name* that propagates taint information, but a class method of the second string of type *str* which does not propagate taint information. As a result, variable *header* is not taint-aware. Therefore, the existing approach can **not** recognize this taint information when it reaches a taint sink and report a security issue.

Consequently, it is evident that we flag a number of False Negatives (FN) in the approach of Conti & Russo. To deal this issue, we introduce a new complementary static taint analysis of our own, the details of which are described in the next section.

```
1  # class 'TaintAwareClass' created by taint source
2  name = request.POST.get("name")
3
4  # class 'TaintAwareClass' is returned
5  header = "Welcome user, " + name
6
7  # class 'TaintAwareClass' reached the taint sink
8  return render(request, 'index.html', {'header':header})
```

Figure 19: Example of taintedeness preservation during dynamic taint analysis

```
# class 'TaintAwareClass' created by taint source
name = request.POST.get("name")

# class 'str' is returned
header = "Welcome user, {}".format(name)

# class 'str' reached the taint sink
return render(request, 'index.html', {'header':header})
```

Figure 20: Example of taintedeness revocation during dynamic taint analysis

To conclude, in figure 21 we present some examples of taintedeness preservation and revocation during dynamic taint analysis:

```
# taint source
name = request.POST.get("name")

# result ok: class of header is 'TaintAwareClass'
header = "Hello user, " + name
header0 = name.upper()

# problem: class of header is 'str'
header1 = "Welcome user, {}".format(name)
header2 = f"Welcome user, {name}"
header3 = "Welcome user, test".replace("test", name)

# taint sink
return render(request, 'index.html', {'header':header})
```

Figure 21: Comparison of taintedeness preservation and revocation during dynamic taint analysis

6.2 Static Taint Analysis theoretical framework

To solve the aforementioned issue, we introduce a complementary customised version of static taint analysis. The idea of a hybrid analysis (dynamic and static) is not new and has been successfully adopted by other authors, in the past [4] [5].

Static taint analysis allows us to learn about program's properties without executing it and, thus provides better code coverage compared to dynamic analysis tools. The goal is to track flow from sources to sinks, whose position is already known from dynamic analysis. The most concerning disadvantage is that static analysis inherently tends to produce some False Positives (FP).

In computability theory, Rice's theorem [7] states that all non-trivial, semantic properties of programs are undecidable. Semantic properties concern the program's behavior, while syntactic properties concern the program's syntax. Also, a property is non-trivial if it is neither true for every computable function, nor false for every computable function. Since we analyze the semantics of a program, we conclude that static analysis is undecidable. Nevertheless, sometimes a loss of precision is necessary to make the semantics decidable. Consequently, we need to make a compromise between precision and decidability. We use approximate decidable answers and we consider the following assumptions:

- We track only assigments and function/method calls in local scope. That means we do not track global, nonlocal or other variables beyond the local scope and we do not inspect the content of a called function.
- If a tainted variable is passed as an argument in an non-sanitizer and non-sink function, we consider the function always returns tainted content.

6.3 Static Taint Analysis implementation

Our tool uses the *ast* module provided by the Python Standard Library to build an Abstract Syntax Tree (AST) [18] of a target code file. ASTs have been used, also, by other authors such as Kamal [6] who used ASTs to analyze static source code for XSS vulnerabilities in ASP.NET web applications.

ASTs represent the source code as a tree data structure and abstract different syntax elements of the source code into separate groups. This allows us to travel through the source code easily and examine it [6].

AST code tree structure consists of multiple nodes. These nodes might contain child nodes. A node consists of statements or expressions [6]. We have enhanced the provided *ast* library by adding a field in every node which points to the parent node. As a result, not only are we able to traverse the AST forwards from root to leafs, but also backwards from leafs to root. This helps us locate variable scopes and traverse the tree quite easily.

Static analysis begins by collecting all the taint source positions that dynamic taint analysis has previously discovered. Positions are in the format file@line. So, we load each target source file from disk and we construct its abstract syntax tree.

We keep a list of tainted and untainted variables, which are updated in every iteration. If a taint sink is reached then we report a flaw only if the variable is included in the tainted variables. Similarly, if a tainted variable is passed in a sanitizer of those we are aware of, then we move it to the untainted variables list.

Firstly, we locate the source line where tainted data appear. Since our assumption is that we track only the local scope of variables, we traverse the ast backwards until we meet a function, class or module definition. We add all the variables in the list of tainted variables and we invoke the tree parser function.

Parser function examines the code line by line until its reaches the end of local scope or a return call. Specifically, if we reach an assignment operation, we check if the right part of the assignment contains tainted variables. If so, then we mark the variables in left part of the assignment as tainted. If not, we continue.

On ther other hand, if we reach a call statement we have to figure out if it is a sanitizer or a taint sink. Sanitizer and taint sink functions are known to static analysis beforehand, as they are the same for static and dynamic taint analysis and have been specified during program's initialization. As a result, we compare the function names against a whitelist of functions and we confirm there is no homonymity from the import statements in the beginning of the source file. For example, if sanitizer *escape* is found, then we expect to find the *from django.utils.html import escape* statement above. For sanitizers, we insert or move the to the untainted variables list all the variables included in the function's arguments. For taint sinks, we check if any tainted variable is included in the function's argument list and if so we report a security issue.

In figure 22 we present an example for which dynamic taint analysis reports no flaw, while such a flow exists. Specifically, as table 2 presents, variable *header* is tainted but not recognized as tainted at runtime, unlike untainted variable *intro* which receives the tainted

variable *username* but it sanitizes its content. Static taint analysis successfully reports a flaw and consequently it offers a complete taint analysis test case.

```
1  name = request.POST.get(”name”)
2
3  name = name.upper()
4  header = f”Dear, {name}”
5  username = ”user_” + name
6  intro = ”Thank you for signing up. You username is: ” + escape(username)
7
8  return render(request, ’signupcomplete.html’, {’header’:header, ’intro’:intro})
```

Figure 22: Example that dynamic analysis reports no flaws

Table 2: Taint status of variable during Dynamic and Static analysis

Line	Variable name	Taint status in dynamic analysis	Taint status in static analysis
3	name	Tainted	Tainted
4	header	**Not tainted**	**Tainted**
5	username	Tainted	Tainted
6	intro	Not tainted	Not tainted

7. EXPERIMENTAL RESULTS

7.1 Test preparation

For the needs of this thesis, we conducted a series of tests. Specifically, we installed a Django server on Ubuntu Linux and run a set of manual tests. We demonstrate that the tool successfully detects vulnerabilities.

For the tool to work, it is required to install *html5lib, bs4, cssutils, pyjsparser* modules. Other modules may be required depending on what's already installed. *Manage.py* has Django's *__main__* function. So, we added the library import statements in the source file in order to initialize our tool simultaneously with Django.

Django is a Model-View-Template (MVT) framework and there is no separate controller for clients requests. All the functionality regarding HTTP request handling is done by the View functions. These are contained in the *views.py* file of the app.

Briefly, when receiving an HTTP request, Django analyzes the route (URL). For example, in a request to *https://www.example.com/myapp/*, it looks for *myapp/*. When Django finds a matching pattern, it calls the specified View function with an HttpRequest object and the parameters from the route as arguments. Each view is responsible for returning an HttpResponse object to client or raising an exception such as HTTP 404 [34].

Consequently, user-provided data that are potentially harmful are processed by View functions. We develop tests cases that concern View functions.

7.2 Testing dynamic taint analysis

7.2.1 Testing scope & implementation details

We conducted a series of manual tests in each phase of development to verify that the tool works. We considered test cases where Views handle HTTP GET and POST requests. Program's behavior with other types of requests such as PUT, PATCH and DELETE is the same, so we were not addressing them.

The goal was to verify if our tool can detect tainted data starting from a taint source and ending up in a taint sink. During information flow, a sanitizer or more could have been applied so we needed to test for which cases our tool successfully detected flaws and for which it failed.

As a result, we considered in each test a set of taint source, taint sink, sanitizer. We combined taint sources, sinks and sanitizers from table 3, selecting for each test a different source, sink and sanitizer to cover as many tests cases as possible. We performed the same tests omitting the sanitizers. In each test we mixed tainted data with untainted data of type *str* to check if the tool reports a flaw (examples in figures 21, 20, 19).

Ragarding the taint sources, sinks and sanitizers we chose to test, we comment the following:

1. **Taint sources**

   Since, we considered test cases where Views handle HTTP GET and POST requests it sufficed to test only those of table 3. The behavior with other types of requestes was expected the same.

2. **Taint sources**

   We tested all the security sensitive functions included in the tool's current settings configuration (table 3). As a result, if a user decides to expand the list of taint sinks, even though we expect no different program behavior, new tests have to be made.

3. **Sanitizers**

We tested all the sanitizer functions included in the tool's current settings configuration (table 3). Respectively, if a user decides to expand the list of sanitizers, even though we expect no different program behavior, new tests have to be made.

Table 3: Taint status of variable during Dynamic and Static analysis

Role	Functions tested
Taint Source	request.POST.get() request.GET.get()
Taint sink for XSS	render() HttpResponse() HttpResponse.write() JsonResponse() StreamingHttpResponse()
Taint sink for SQL injection	Manager.raw() Manager.extra() Manager.annotate() Cursor.execute()
Taint sanitizers	django.utils.html.escape xmlrpc.client.escape xml.sax.saxutils.escape html.escape django.utils.html.escapejs urllib.parse.urlencode django.utils.http.urlencode soupsieve.css_parser.escape

7.2.2 Presenting a test case

In the following figures, we present an example of dynamic taint analysis flaw report. We have a View function that handles the contact form of a website:

```
class ContactForm(forms.Form):
    from_email = forms.EmailField(label='Enter email')
    subject = forms.CharField(label='Enter subject', max_length=30)
    message = forms.CharField(label='Enter message', widget=forms.Textarea)


def contact(request):
    if request.method == 'POST':
        from_email = request.POST.get("from_email")
        subject = request.POST.get("subject")
        message = request.POST.get("message")
        #..store data in database
        return render(request, 'thanks.html', {'from_email': from_email,
                                               'subject': subject, 'message': message})
    elif request.method == 'GET':
        form = ContactForm()
        return render(request, 'contact.html', {'form': form})
    else:
        return HttpResponse("Method not supported")
```

Figure 23: Code of a Django View function for test case 1

We request the contact form in the client with a GET request. So, Django runs the code in figure 23 that corresponds to GET request handling, it renders the following template and sends it to the client:

```
<form action="/signup/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>
```

Figure 24: Code of a Django template that renders a contact form for test case 1

From client, we fill in the following form and we send it to the server with a POST request:

Figure 25: Browser page of the rendered template for test case 1

Django processes used-provided data in view of figure 23 and renders the following template with a response:

```
<html>
    <p> Thanks for contacting! Message:</p>
    <p id="message"></p>
    <script>
        var par, query;
        par = document.getElementById ("message");
        query = "{{ message }}";
        par.innerHTML = query;
    </script>
    </body>
</html>
```

Figure 26: Code of a Django template that renders server response for test case 1

In browser, we see this:

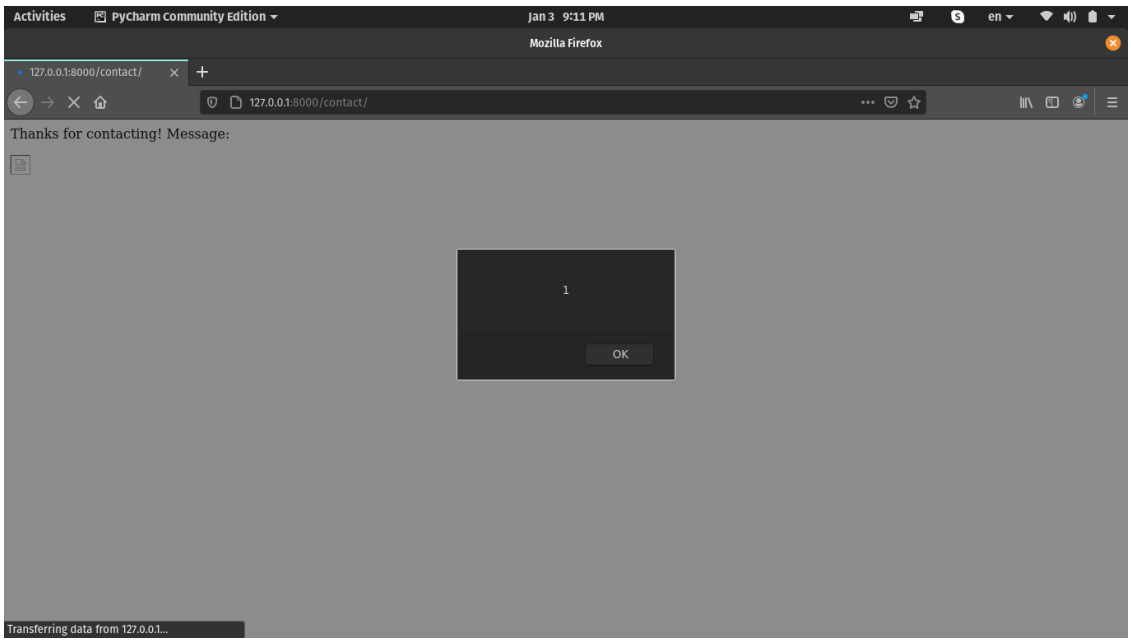

Figure 27: Browser page with successful XSS attack for test case 1

That means there is a successful XSS attack. Our tool reports the flaw with the use of dynamic taint analysis:

```
Django version 3.1.2, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
DYNAMIC ANALYSIS REPORT
Cross-Site Scripting (XSS) flaw found
-------------------------------------
Detected browser context sequence: ['e', 'H']
Sanitizer sequence: ['a', 'q', 'r']
-------------------------------------
Flaw position: b'/home/giorgos/PycharmProjects/pythonProject-mine/newsletter/views.py'@98
-------------------------------------
Position details:

[03/Jan/2021 19:11:38] "POST /contact/ HTTP/1.1" 200 305
            subject = request.POST.get("subject")
            message = request.POST.get("message")
            #..store data in database
-->         return render(request, 'thanks.html', {'from_email': from_email,
                                                   'subject': subject, 'message': message})
        elif request.method == 'GET':

[03/Jan/2021 19:11:38] "GET /contact/N/A HTTP/1.1" 200 666
```

Figure 28: Flaw report generated by our tool for test case 1

7.3 Testing server-side parsing

Testing server-side parsing and especially Model Browser & Sanitization Verifier modules is easy. Specifically, we conducted a series of manual tests. Since parsing concerns only the Django Template Language (DTL) responses, we created templates in our Django application. Templates containted HTML, CSS and URLs with *data:* and *javascript:* keywords. After, we run the server and printed the found browser context sequence. We manually verified that the browser context results were the expected.

7.4 Testing static taint analysis

To verify that static taint analysis worked we applied the same steps as in dynamic taint analysis. We present an example where dynamic taint analysis failed, while static taint analysis succeeded. We have a View function that handles a whitepages service and looks up phone numbers in the whitepages database:

```
def phone_lookup(request):
    if request.method == 'GET':
        phone = request.GET.get('phone')
        # .. search for phone number in whitepages database
        result = "You searched for phone number: {}".format(phone)
        return render(request, 'whitepages.html', {'result': result})
    else:
        return HttpResponse("Method not supported")
```

Figure 29: Code of a Django View function for test case 2

In the website we fill in the phone number we are searching for. Client adds the phone number as a parameter in an HTTP GET request and sends it to Django. But, instead of a phone number we manually enter *\x3cimg src=\x22N/A\x22 onerror=\x22alert(1) \x22 /\x3e* which translates to **. Django will process the request in the view function of figure 29 and render the following template response:

```
<!DOCTYPE html>
<html>
    <body>
        <p id="message"></p>
        <script>
            var par, query;
            par = document.getElementById ("message");
            query = "{{ result }}";
            par.innerHTML = query;
        </script>
    </body>
</html>
```

Figure 30: Code of a Django template that renders server response for test case 2

In browser, we see this:

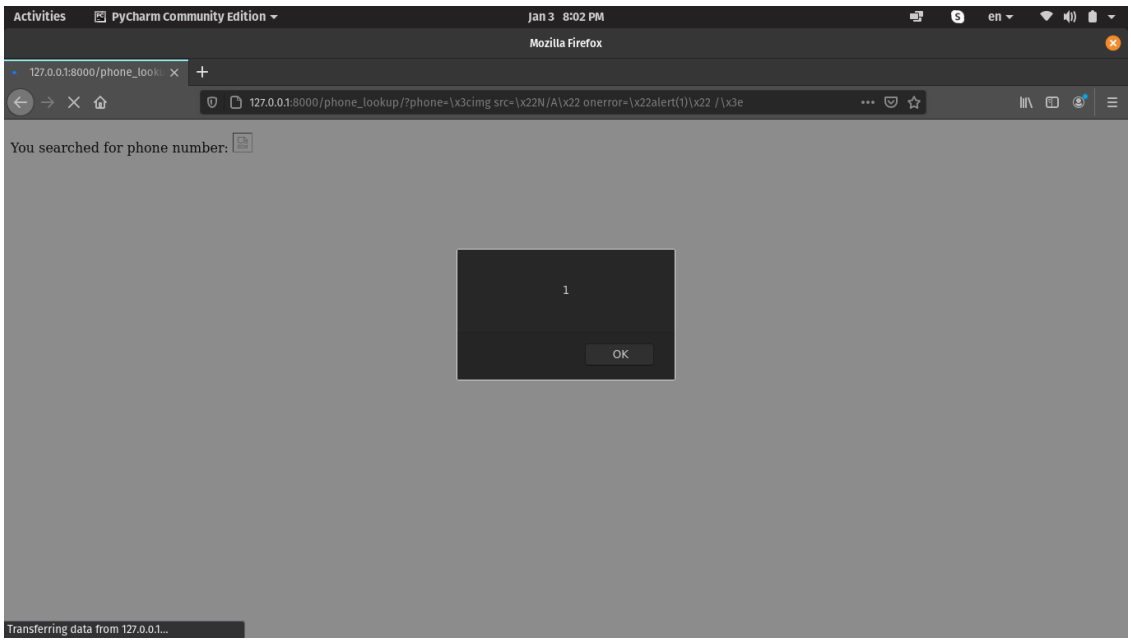

Figure 31: Browser page with successful XSS attack for test case 2

That means that the XSS attack has been successful. In the following figure, we see that static taint analysis has reported the flaw, while dynamic taint analysis has not.

```
Django version 3.1.2, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
STATIC ANALYSIS REPORT
Potential Cross-Site Scripting (XSS) found
--------------------------------------
It is suggested the user verifies the validity of the report
--------------------------------------
Taint Source (Flaw position): /home/giorgos/PycharmProjects/pythonProject-mine/newsletter/views.py@104
Taint Sink (Flaw position): /home/giorgos/PycharmProjects/pythonProject-mine/newsletter/views.py107
--------------------------------------
Taint Sink details:
render(request, 'whitepages.html', {'result': result})

[03/Jan/2021 17:59:53] "GET /phone_lookup/?phone=\x3cimg%20src=\x22N/A\x22%20onerror=\x22alert(1)\x22%20/\x3e HTTP/1.1" 200 384
```

Figure 32: Flaw report generated by our tool for test case 2

7.5 Testing results

Apparently, testing worked and, therefore, gave us the opportunity to discover where the approach of Conti & Russo and Steinhauser & Tůma lacks. We were able to identify that taintedeness of data can be revoked as tainted information flows from a taint source to a taint sink and why that happens. Thus, we were able to study the nature of False Negatives (FP) in dynamic taint analysis and overcome this situation applying the complementary static taint analysis. Of course, the test suite can be further improved, but until now it has been proven useful.

8. CONCLUSION

8.1 Summary

In this thesis, we implemented an execution monitor which combines hybrid taint analysis and server-side parsing in order to discover context-sensitive XSS flaws in template rendering, context-insensitive XSS flaws in simple HTTP responses and potential SQL injection in raw SQL queries in Django web applications.

We provided an analysis tool via a library entirely written in Python based on the approach presented by Conti & Russo [2], later adopted by Steinhauser & Tůma [1]. To intercept the taint sources, taint sinks, sanitizers and parsers, we conscripted Python decorators and used server-side parsing to determine the browser context of tainted values and validate the results.

Our approach can successfully discover context-aware XSS flaws and SQL injection without modifications of Python interpreter. Consequently, it is presented as a flexible and portable solution that can be attached to any Django web application.

We apply dynamic taint analysis to monitor the information flow during execution time and record all sanitizers applied. If tainted data are passed to a template then we user server-side parsing to determine the browser context of tainted values and validate the results. Else, if tainted data have reached a simple HTTP response or a raw SQL query we report a flaw immediately.

During dynamic analysis deployment we discovered that, in some cases, taintedeness of data can revoked as tainted information flows from a taint source to a taint sink. As a result, untainted data would reach the taint sink and, therefore, no flaw would be reported even though it would exist.

To tackle the problem, we introduced a new customised version of static taint analysis that builds an Abstract Syntax Tree (AST) of a target code file and parses it line by line. We used approximate decidable answers to tackle the undecidability problem.

In conclusion, our implementation offers a extended set of analysis procedures for vulnerability detection in Django web applications and provides the potential to be further extended in the domain of dynamic & static taint analysis.

8.2 Future work

In this section we provide guidlines for future work. Undoubtedly there are more aspects of this approach that could be developed further. Specifically we focus on dynamic and static analysis.

On the one hand, as discussed above, detection of nested context in JavaScript strings is an algorithmically undecidable problem. JavaScript is a Turing-complete language that can perform arbitrary operations with the tainted string comparing its definition with its usage. The same string might even be cloned and end up in different context combinations. One possible solution, suggested as future work, would be the extension of data flow analysis inside the JavaScript code.

On the other hand, static analysis is undecidable as a conclusion from Rice’s theorem. In this thesis, we make a compromise between precision and decidability. In the future, evolvement could be possible.

TERMINOLOGY TABLE

Αυτο-απολύμανση χωρίς ευαισθησία περιβάλλοντος	Context-insensitive auto-sanitization
Δυναμική Ανάλυση	Dynamic Taint Analysis
Στατική Ανάλυση	Static Taint Analysis
Υβριδική Ανάλυση	Hybrid Analysis
Συντακτική Ανάλυση πλευράς Διακομιστή	Server-side parsing
Πηγή εισαγωγής ευαίσθητων δεδομένων	Taint source
Καταβόθρα κατάληξης δεδομένων	Taint sink
Απολυμαντής	Sanitizer
Συντακτικός αναλυτής	Parser
Ακατέργαστο επερώτημα SQL	Raw SQL query
Προσημείωση μολυσματικότητας δεδομένων	Taintedeness
Ευπάθεια XSS με ευαισθησία περιβάλλοντος	Context-sensitive XSS flaw
Ευπάθεια XSS χωρίς ευαισθησία περιβάλλοντος	Context-insensitive XSS flaw
Μολυσμένος	Tainted
Μη μολυσμένος	Untainted

ABBREVIATIONS, ACRONYMS

AST	Abstract Syntax Tree
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
ORM	Object-Relational Mapping
SQL	Structured Query Language
URI	Uniform Resource Locator
W3C	World Wide Web Consortium
XSS	Cross-site Scripting

REFERENCES

[1] Steinhauser A, Tůma, P. DjangoChecker: Applying extended taint tracking and server side parsing for detection of context-sensitive XSS flaws. Softw: Pract Exper. 2019; 49: 130-148. [Accessed: Dec 4, 2020].

[2] Conti JJ, Russo A. A taint mode for python via a library. In: Proceedings of the 15th Nordic Conference on Information Security Technology for Applications; 2012; Espoo, Finland. [Accessed: Dec 4, 2020].

[3] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. 2011. A systematic analysis of XSS sanitization in web application frameworks. In Proceedings of the 16th European conference on Research in computer security (ESORICS'11). Springer-Verlag, Berlin, Heidelberg, 150–171.

[4] Vogt P, Nentwich F, Jovanovic N, Kirda E, Kruegel C, Vigna G. Cross site scripting prevention with dynamic data tainting and static analysis. Paper presented at: International Symposium on Network and Distributed System Security; 2007; San Diego, CA. [Accessed: Dec 4, 2020].

[5] Lam MS, Martin M, Livshits B, Whaley J. Securing web applications with static and dynamic information flow tracking. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation; 2008; San Francisco, CA. [Accessed: Dec 4, 2020].

[6] Mahendra, Kamal. (2020). Develop a set of guidelines, static code analysis (Taint Analysis) to prevent XSS attacks in ASP.NET web applications.

[7] H. G. Rice, Classes of Recursively Enumerable Sets and Their Decision Problems, Transactions of the American Mathematical Society Vol. 74, No. 2 (Mar., 1953), pp. 358-366. [Accessed: Dec 4, 2020].

[8] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. 2009 Marktoberdorf Summer School (IOS Press), 2009. [Accessed: Dec 4, 2020].

[9] "Microsoft SQL Injection" [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms161953(v=sql.105)?redirectedfrom=MSDN. [Accessed: Dec 5, 2020].

[10] "OWASP Top 10 - 2017" [Online]. Available: https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%20Top%2010-2017%20(en).pdf. [Accessed: Dec 5, 2020].

[11] "OWASP Top 10 - 2010" [Online]. Available: https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2010.pdf. [Accessed: Dec 5, 2020].

[12] "XSS Prevention Cheat Sheet" [Online]. Available: http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet. [Accessed: Dec 5, 2020].

[13] "Guardian: TalkTalk hit with record £400k fine over cyber-attack" [Online]. Available: https://www.theguardian.com/business/2016/oct/05/talktalk-hit-with-record-400k-fine-over-cyber-attack. [Accessed: Dec 5, 2020].

[14] "BBC: US man stole 130m card numbers" [Online]. Available: http://news.bbc.co.uk/2/hi/americas/8206305.stm. [Accessed: Dec 5, 2020].

[15] "ZDNet: 450,000 user passwords leaked in Yahoo breach" [Online]. Available: https://www.zdnet.com/article/450000-user-passwords-leaked-in-yahoo-breach/. [Accessed: Dec 5, 2020].

[16] "The 15 Biggest Data Breaches in the Last 15 Years" [Online]. Available: https://www.visualcapitalist.com/the-15-biggest-data-breaches-in-the-last-15-years/. [Accessed: Dec 5, 2020].

[17] "The Map of Cybersecurity Domains (version 2.0)" [Online]. Available: https://www.linkedin.com/pulse/map-cybersecurity-domains-version-20-henry-jiang-ciso-cissp. [Accessed: Dec 5, 2020].

[18] "Abstract Syntax Trees library in Python Standard Library" [Online]. Available: https://docs.python.org/3/library/ast.html. [Accessed: Dec 4, 2020].

[19] "Html5lib: A pure-python library for parsing HTML" [Online]. Available: https://pypi.org/project/html5lib/. [Accessed: Dec 5, 2020].

[20] "Cssutils: A Python package to parse and build CSS Cascading Style Sheets" [Online]. Available: https://pypi.org/project/cssutils/. [Accessed: Dec 5, 2020].

[21] "Cssutils Documentation" [Online]. Available: https://pythonhosted.org/cssutils/docs/css.html. [Accessed: Dec 5, 2020].

[22] "PyJsParser: Fast JavaScript parser" [Online]. Available: https://github.com/PiotrDabkowski/pyjsparser. [Accessed: Dec 5, 2020].

[23] "Beautifulsoup4: A Python library designed for quick turnaround projects like screen-scraping" [Online]. Available: https://pypi.org/project/beautifulsoup4/. [Accessed: Dec 5, 2020].

[24] "Beautiful Soup Documentation" [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/. [Accessed: Dec 5, 2020].

[25] "W3C Recommendation 3 October 2017 on HTML 5.1, 2nd Edition" [Online]. Available:

https://www.w3.org/TR/html51/syntax.html#writing-html-documents-elements. [Accessed: Dec 5, 2020].
[26] ”JavaScript Grammar and Types” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types. [Accessed: Dec 5, 2020].
[27] ”IETF RFC 2397: The ”data” URL scheme” [Online]. Available: https://tools.ietf.org/html/rfc2397. [Accessed: Dec 5, 2020].
[28] ”The At-Rules of CSS” [Online]. Available: https://css-tricks.com/the-at-rules-of-css/. [Accessed: Dec 5, 2020].
[29] ”Understanding CSS Syntax” [Online]. Available: https://css-tricks.com/the-at-rules-of-css/. [Accessed: Dec 5, 2020].
[30] ”OWASP Types of XSS” [Online]. Available: https://owasp.org/www-community/Types_of_Cross-Site_Scripting. [Accessed: Dec 5, 2020].
[31] ”OWASP DOM Based XSS” [Online]. Available: https://owasp.org/www-community/attacks/DOM_Based_XSS. [Accessed: Dec 5, 2020].
[32] ”Django Documentation: Security in Django” [Online]. Available: https://docs.djangoproject.com/en/3.1/topics/security/. [Accessed: Dec 5, 2020].
[33] ”Django Documentation: Performing raw SQL queries” [Online]. Available: https://docs.djangoproject.com/en/3.1/topics/db/sql/. [Accessed: Dec 5, 2020].
[34] ”Django Documentation: Writing views” [Online]. Available: https://docs.djangoproject.com/en/3.1/topics/http/views/. [Accessed: Dec 5, 2020].
[35] ”Top 10 Django Apps” [Online]. Available: https://www.netguru.com/blog/django-apps. [Accessed: Dec 5, 2020].