UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

VULNERABILITIES DETECTION AT RUNTIME AND CONTINUOUS AUDITING

MESTRADO EM SEGURANÇA INFORMÁTICA

Bruno Octávio Horta Lourenço

Dissertação orientada por:
Prof. Doutora Ibéria Vitória de Sousa Medeiros
e co-orientado pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

2020

Acknowledgments

First I would like to express my gratitude to Professor Ibéria Medeiros for guiding this project by the relevant advice and observations that have been given, and for always being available to help and contribute to this project success.

To Professor Nuno Neves, for his co-orientation, I thank the availability and trust provided unconditionally, which contributed decisively to my success.

A huge thanks to my family, and especially to my wife Cristina and daughter Alice, who form a strong foundation to reach my goals through love, strength, patience, support, determination, and encouragement.

Finally, I would like to thank my parents who gave me support and availability that greatly contributed to this work.

This work was partially supported by the national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference to PTDC/CCI-INF/29058/2017, project SEAL, and LASIGE Research Unit (UIDB/00408/2020).

I dedicate this work to the love of my life, my wife Cristina, a companion of all hours, who contributed decisively to the conclusion of this dissertation, with always pertinent suggestions. To my daughter Alice, who gave a special meaning to my existence and has given me great moments and happiness.

Resumo

Na atualidade, a integração de funcionalidade e segurança em aplicações é um desafio. Existe a noção de que a segurança é um processo pesado, requer conhecimento e consome o tempo dos programadores, contrastando desta forma com a visão relativa à funcionalidade. Independentemente destes desafios, é importante que as organizações tratem da segurança nos seus processos ágeis, pois os ativos críticos da organização devem ser protegidos contra potenciais ataques. Uma forma de evitar que os ataques tenham sucesso passa por integrar ferramentas que possam ajudar a identificar vulnerabilidades de segurança durante a fase de desenvolvimento das aplicações e sugerir métodos para a sua correção.

Segundo o Instituto Gartner, mais de 75% dos problemas com segurança na Internet são devidos a vulnerabilidades exploráveis a partir das Aplicações Web (*Web Apps*). A maior parte das *Web Apps* são naturalmente vulneráveis devido às tecnologias adotadas na sua concepção, à forma como são desenhadas e desenvolvidas, e ao uso de vários objetos e recursos, além da integração de outros sistemas. Frequentemente observa-se que são priorizados os aspetos funcionais que atendem a área de negócios, enquanto os requisitos de segurança ficam em segundo plano.

Os ataques a *Web Apps* podem causar problemas de variados níveis de impacto, como por exemplo: interrupção ou queda de desempenho do serviço; acesso não autorizado a dados confidenciais e estratégicos; roubo de informação e clientes; fraudes e modificação de dados no fluxo das operações; perdas financeiras diretas e indiretas; prejuízos à imagem da marca da empresa; perda da lealdade dos clientes e gastos extraordinários com incidentes de segurança. Os riscos de ataques mais comuns são genericamente conhecidos e podem ser previstos com antecedência, pois são listados pela *Open Web Application Security Project* (OWASP), e dentre eles, três dos principais são: *SQL Injection* (SQLi); *Cross-Site Scripting* (XSS); *Broken Authentication* e *Session Management*. Os ataques mais graves são aqueles que, quando realizados sobre vulnerabilidades da *Web App*, não serão detetados de imediato e resultam no acesso a dados sigilosos do negócio, da infraestrutura, ou de clientes, e que podem ser posteriormente organizados para realizar um ataque de impacto mais relevante, ou uma fraude. Neste contexto, um novo paradigma surge no que se refere à auditoria em ambientes web.

O conceito de Auditoria Contínua (AC) emerge como uma nova solução de auditoria que responde a novas necessidades, sendo um tema recente que tem sido objeto de pesquisas e aposta de organizações. O modelo tradicional de auditoria, baseado em análises pontuais e descontínuas, torna-se cada vez mais inadequado à dinâmica atual da informação e aos sistemas que a gerem. Atualizações constantes de aplicações e as alterações nas configurações do sistema podem introduzir vulnerabilidades e deixar uma organização suscetível a ataques. Portanto, para manter os dados seguros, os sistemas e dispositivos devem ser verificados continuamente para identificar e relatar vulnerabilidades à medida que são descobertas. Este conceito traduz-se numa enorme mudança na filosofia tradicional da auditoria para um paradigma de AC que torna possível uma intervenção e ação corretiva mais cedo. Desta forma, é necessário que as organizações adotem uma metodologia que permita aos auditores independentes, fornecer garantias por meio de relatórios sobre a ocorrência de eventos ao longo da vida do sistema. Esses eventos, quando monitorizados em tempo real, permitem desvios a serem detetados e relatados para aumentar a velocidade e a eficácia da resposta pelos elementos responsáveis pela tomada de decisão.

As organizações estão sujeitas a vários tipos de auditorias que têm diferentes finalidades, como a qualidade, o ambiente, a operação ou a gestão. Estes processos seguem um período de tempo para validar e analisar o que já foi feito e o estado atual da organização. Na segurança da informação, a AC visa garantir a monitorização em tempo real do sistema e o risco dos ativos da empresa. Para além disso, permite avaliar o nível de segurança atual do sistema, monitorizar o sistema em tempo real, aumentando a eficiência da descoberta e mitigação de vulnerabilidades. Os testes de intrusão, são geralmente um complemento para a AC. Num processo contínuo em que não existe esse comportamento invasivo, as análises de vulnerabilidades são realizadas com o auxílio de ferramentas automáticas ao longo do tempo para observar e monitorizar o estado do sistema e as ações corretivas as serem tomadas.

O objetivo desta tese é propor uma abordagem e desenvolver uma ferramenta que permitirá detetar ataques do tipo *Injection Attacks* (IA) ou *Cross-Site Request Forgery* (CSRF) em *Web Apps*, no caso de estas estarem a recorrer ao mecanismo *Cross-Origin Resource Sharing* (CORS). Para efetuar a deteção de IA, a ferramenta terá a capacidade de analisar os links externos que são passados no atributo `href` a que uma *Web App* se liga, com o intuito de verificar se estes estão comprometidos. Para a deteção de CORS a ferramenta analisará todos os links internos passados no atributo `src` para verificar se estes invocam métodos `XMLHttpRequest` utilizados para chamadas de CORS. Estes dois tipos de ataques estão sempre associados, contribuindo para um IA bem-sucedido. O IA é uma classe de ataques que depende da injeção de dados numa *Web App*, causando a execução ou interpretação de dados mal-intencionados de maneira inesperada. Exemplos de ataques desta classe incluem *SQLi*, *HTML Injection*, *XSS*, *Header Injection*, *Log Injection* e *Full Path Disclosure*. Estes são os ataques mais comuns e bem-sucedidos na

Internet devido aos seus numerosos tipos, grande superfície de ataque e complexidade necessária para os proteger.

O CORS é um mecanismo do browser que permite o acesso controlado a recursos localizados fora de um determinado domínio. Ele estende e adiciona flexibilidade à *Same Origin Policy* (SOP). No entanto, este mecanismo também oferece potencial para ataques baseados em vários domínios, se a política de CORS de um site estiver mal configurada ou implementada. O CORS não pretende ser uma proteção contra ataques de *Cross-Request* como o CSRF.

Tendo em conta o anteriormente descrito relativamente a IA e CORS, a ferramenta desenvolvida permite a deteção de vulnerabilidades em *Web Apps* em AC. O foco fundamental está nos links externos e internos da *Web App*. Corre num servidor web, disponibilizando este serviço aos utilizadores na internet, permitindo analisar ligações externas e internas de uma determinada *Web App*. Para as ligações externas irá detetar evidências de IA, atribuindo uma classificação de benigno ou maligno às ligações externas identificadas. Para os links internos, verifica se existem chamadas de *Cross-Origin* mais especificamente CORS. Desta forma um utilizador poderá submeter o URL da sua *Web App* que irá ser analisado pela ferramenta Vulnerabilities Detector at Runtime and Continuous Auditing (VuDRuCA) que recorre a um mecanismo de AC.

A ferramenta VuDRuCA emprega técnicas de *crawling* para navegar nas páginas da *Web App* e obter a informação pretendida. Utiliza ainda a API do Virus Total para analisar URLs, identificando conteúdo malicioso detetável por antivírus e scanners de Web Apps. Como *backend* a ferramenta utiliza uma base de dados relacional que armazena todos os dados recolhidos para que estes possam ser analisados, contribuindo para a apresentação de indicadores.

Na fase de avaliação a ferramenta foi testada utilizando uma amostragem de 100 URLs de *Web App* que recorrem à tecnologia AJAX. Para estes foram contabilizados o número de sites externos e internos da *Web App*. Após uma primeira análise foram escolhidos 30 *Web Apps* para categorização, medição dos tempos de execução para deteção de links externos e internos e várias outras métricas relativas aos tempos de execução. Finalmente para testar o motor de AC foram selecionados 10 URL de *Web Apps* que na sua maioria recorrem a CORS. Nestas 10 *Web Apps* foi identificada a tecnologia de *Content Manamgment System* (CMS) utilizada. O módulo de AC, efetuou ainda uma análise durante um período de 5 dias, com intervalos de 24h, para validar se existia a introdução de novos links externos ou se algum destes estava comprometido. Relativamente aos links internos foi validado se existiam novos links internos e se estes recorriam a CORS.

Palavras-chave: vulnerabilidades, aplicações web, auditoria contínua, auditoria estática de código, segurança de software.

Abstract

Nowadays integrating applications agility and security is an extremely challenging process. There is the notion that security is a heavy process, requiring knowledge and consuming time of the development teams. On the other hand, the acquisition of Web Applications (Web Apps) is often achieved through contracted services because companies do not have the necessary software developers. Taking this fact into account, the risk of obtaining a product implemented by poorly qualified developers is a reality.

The main objective of this thesis is to propose a solution and develop a tool that will detect some forms of Injection Attacks (IA) or Cross-Site Request Forgery (CSRF) attacks in Web Apps. The latter is due to the fact that Web Apps sometimes employ Cross-Origin Resource Sharing (CORS). Some statistics demonstrate that these attacks are some of the most common security risks in Web Apps. IA is a class of attacks that relies on inputting data into a Web App to make it execute or interpret malicious information unexpectedly. Examples of attacks in this class include SQL Injection (SQLi), Header Injection, Log Injection, and Full Path Disclosure.

CORS is used by browsers to allow controlled access to resources located outside a given domain. It extends and adds flexibility to the Same Origin Policy (SOP). However, this mechanism also offers the potential for Cross-Domain based attacks if a site's CORS policy is misconfigured. CORS is not intended to be a protection against Cross-Request attacks like the CSRF.

The developed tool, called VuDRuCA, allows the detection of vulnerabilities associated with IA and CORS in Web Apps. It runs on a web server, providing this service to users on the internet, allowing them to analyse external and internal links of a particular Web App. For the external links, it will detect evidence of IA, assigning a benign or a malign classification to the identified external links. For internal links, there is a check for Cross-Origin calls, specifically CORS.

VuDRuCA uses crawling techniques to navigate through the pages of the Web App and obtain the desired information. It also uses the Virus Total API, which is a free online service that parses URLs, enabling the discovery of malicious content detectable by antivirus and website scanners. As a backend, it uses a relational database to store the collected data so that it can be retrieved and analysed, reporting the presence of security indicators.

Keywords: vulnerabilities, web applications, continuous auditing, static code auditing, software security.

Contents

List of Figures

Chapter 1

Introduction

Nowadays, the web is the most relevant and powerful platform for all new software applications. As a result, new Web Applications (Web Apps) are constantly being developed, making the security of such applications become increasingly important. In parallel, the number of reported Web Apps vulnerabilities grows every year, whether they are published in specialized databases (e.g., Common Vulnerabilities and Exposures (CVE)) or discovered internally in the organizations. These vulnerabilities can pose a serious risk of exploitation and may result in system compromise, information leaks, or denial of service. But more serious than this is that vulnerabilities in Web Apps can prove costly for organizations. These costs may include direct financial losses, increases in required technical support and tarnished image and brand.

The Open Web Application Security Project (OWASP) is an example of an open community that is dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted. The OWASP Top Ten lists the 10 most dangerous current Web Apps security flaws, along with effective methods of dealing with them [8]. Project members include a variety of security experts from around the world who share their knowledge of vulnerabilities, threats, attacks, and countermeasures. But the main issue is, how should software engineers develop secure Web Apps? Different developers have diverse opinions regarding which language, framework, or vulnerability-finding tool tends to yield more secure software. For example, the choice of the programming language has an important influence on the security of the Web Apps as the offered constructs may facilitate or prevent certain classes of attacks. In any case, as it is always possible to introduce bugs in the applications, it is necessary to select the most effective testing methods. This could help to reduce risk and allocate resources more appropriately.

1.1 Motivation

Web Apps are increasingly used to provide services available on the Web. As they are developed and integrate various technologies, new types of vulnerabilities have been observed. Also, the number of languages that are used to develop Web Apps can open novel attack vectors for malicious actors and new vulnerabilities.

Currently, 25.7% of the vulnerabilities in CVE are classified as either SQL Injections (SQLi) or Cross-Site Scripting (XSS) [43], occurring most of the time in Web Apps. Briefly, SQLi makes it possible to rewrite a query made by a Web App to a database, which could create unexpected behavior and result in either data loss or disclosure (e.g., by leaking user names and associated passwords). In contrast, a XSS vulnerability allows attackers to inject malicious code into the client part of the Web App and thereby change the behavior of the code executing in the browser, potentially leaking authorization and/or private information.

Another challenge is related to the fact that HTML5 is an emerging stack for next-generation applications. It is enhancing browser capabilities and enabling the execution of Rich Internet Applications in the context of modern browser architectures. Interestingly HTML5 can run on mobile devices making it even more complicated. HTML5 supports a combination of various components like XMLHttpRequest (XHR), Document Object Model (DOM), Cross-Origin Resource Sharing (CORS) and enhanced HTML/Browser rendering. It also brings several new mechanisms to the browser which were not seen before, like local storage, web SQL, WebSocket, web workers, enhanced XHR and DOM-based XPATH to name a few. Consequently, HTML5 has an enhanced attack surface and points of exploitation for attackers. By leveraging these new mechanisms, malicious actors can craft stealth attacks and silent exploits which are hard to detect and have a significant impact [6].

In this way, it is increasingly important to have tools that identify vulnerabilities in different languages of web programming before an application is put into production or even when it is already deployed. We consider that Continuous Auditing (CA) is a must-have, which will allow continuous monitoring of the systems and assessing the risks to the company's assets, increasing the efficiency of vulnerability discovery and mitigation.

The main objective of this dissertation is to propose a solution and develop a tool named VuDRuCA (Vulnerabilities Detector at Runtime and Continuous Auditing) allowing the detection of vulnerabilities in a Web App in CA. It enables the discovery of vulnerabilities associated with Injection Attacks (IA) and the CORS mechanisms. The tool runs on a web server, providing this service to users, allowing the analysis of the external and internal links of a particular Web App. For the external links, it will detect evidences of IA, assigning a classification of benign or malignant to the identified external links. For the internal links, it checks for Cross-Origin calls, related to CORS. This way, a user can submit his Web App URL that will be analised by VuDRuCA.

1.2 Objectives

This thesis has two objectives. The first objective is to study ways to identify and detect possible indicators of compromise in Web Apps and verify if those are using Cross-Domain requests. For that purpose, we will present a methodology to analyse client-side code returned to the user, performing an HTML and JavaScript (JS) continuous code analysis to detect possible compromised Web Apps or Cross-Domain calls.

The second is to develop a solution and its implementation in a tool for using the CA approach. The tool will use crawling techniques and a commercial external entity called Virus Total (VT) to analyse the external links. For the internal links, the tool will use an internal routine to find CORS calls used to send XHR that is mostly implemented in JavaScript programming language. It is used to send HTTP or HTTPS requests directly to a web server and load the server's response data directly back into the script.

1.3 Contribution

The main contributions of the thesis are:

- A study of web vulnerabilities, especially the ones associated with CORS; in addition, CA is considered to understand how to monitor a Web App over a time period;
- An architecture to continuously audit Web Apps, analysing their external links for malware related problems and internal links against CORS. The VuDRuCA tool implements the architecture;
- Carry out an assessment of the developed tool, namely capturing execution times for site analysis alone or through continuous auditing; the number of external links detected per application as well as their classification; the number of internal links per Web App and invoked CORS calls.

1.4 Thesis Structure

This thesis is organized as follows:

- Chapter 2 briefly explains some relevant concepts and provides a fundamental context for the work;
- Chapter 3 presents the proposed architecture, describing the main components as well;
- Chapter 4 discusses the current implementation of the tool VuDRuCA, explaining in more detail each module;

- Chapter 5 evaluates and validates the tool;
- Finally, Chapter 6 provides conclusions of the developed research and discusses future work that can be built upon the base architecture.

Chapter 2

Context and Related Work

This chapter describes a few classes of web vulnerabilities and presents and discusses some related work that is relevant in this context. The chapter is structured in the following sections:

- Section 2.1 looks into several vulnerability classes that are referenced by OWASP as some of the most critical for Web App;
- Section 2.2 details various tools and techniques used in static analysis;
- Section 2.3 looks into fuzzing techniques and fuzzers that implement each technique;
- Section 2.4 reports some knowledge about symbolic execution;
- Section 2.5 exposes related work about oracles.

2.1 Vulnerabilities

In a general way, security vulnerabilities are bugs that were accidentally introduced during software development. It is important to refer that not all bugs are vulnerabilities, only those that might be exploited or used to compromise the system can be considered a vulnerability. The Microsoft Security Response Center (MSRC) defines a security vulnerability as a weakness in a product that could allow an attacker to compromise the *integrity*, *availability*, or *confidentiality* of that product [40].

- *Integrity* refers to the trustworthiness of a resource. An attacker that exploits a weakness to modify data silently and without authorization is compromising the integrity of a product;
- *Availability* refers to the possibility to access a resource. An attacker that exploits a weakness in a product, denying appropriate user access to it, is compromising the availability;

- *Confidentiality* refers to limiting access to information in a resource in order to avoid disclosure to unauthorized parties. An attacker that exploits a weakness in a product to access non-public information is compromising the confidentiality of that product.

For our work, the fundamental reference for Web App vulnerabilities is the ”OWASP Top 10 - The Ten Most Critical Web Applications Security Risks” [8]. In Section 3.2, we will focus on a special class of vulnerabilities that are becoming more relevant as it is exposed to the increasing use of different technologies and programming languages in Web Apps.

Before we talk about classes of vulnerabilities, we will address the types of client and server-side attacks. Attacks targeted at individual client computers are called *client-side attacks*. These are usually directed at web browsers and instant-messaging applications. *Client-side* attacks are a major font for attackers today. As network administrators and software developers fortify the perimeter, attackers need to find a way to make the victims open the door for them to get into the network.

These attacks target vulnerabilities in client applications that interact with a malicious server or process malicious data. Here, the client initiates the connection that could result in an attack. To achieve this, hacker entices users to click a link, open a document, or somehow get to a website controlled by a malicious entity. If a user does not interact with a server, there is no risk because the client does not process any potentially harmful data sent from the server. A typical example of a *client-side* attack is a malicious web page targeting a specific browser vulnerability that would give the malicious server complete control over the client system. Saxena et al. [48] explains that the complexity of the client-side components of Web Apps has exploded with the increase in popularity of web 2.0 applications. Nowadays, traditional desktop applications, such as document viewers, presentation tools and chat applications are commonly available as online JavaScript (JS) applications. The authors present the term *client-side validation (CSV) vulnerabilities* as new vulnerability class. A typical Web 2.0 application has two parts: a server-side component and a client-side component. The server-side component processes the user's request and generates an HTML response that is sent back to the browser. The client-side code of the Web App, typically written in JS, receives the HTML response from the server. The client-side component executes in the web browser and is responsible for processing input data and dynamically updating the view of the web.

CSV vulnerabilities belong to the general class of input validation vulnerabilities but are different from traditional web vulnerabilities like SQL injection (SQLi) and Cross-Site Scripting (XSS) (see later parts of the section). For example, one type of CSV vulnerability involves data that enters the application through the browser's cross-window communication abstractions and is processed completely by JS code, without ever being sent back to the webserver. Another type occurs when a Web App sanitizes input data before embedding it in its initial HTML response but does not sanitize the data completely and allows for its use in the JS component.

In a *server-side* attack, an attacker submits a malicious input that gets executed on the server. This malicious input could be submitted in many different ways, including web form elements and URL parameters. Since the malicious input needs to be crafted and submitted in a manner that forces the back-end applications on the server to process it, the attacker usually needs to gain an understanding of the back-end applications. This knowledge can be gained for instance by using an information disclosure attack. In any case, depending on the kind of back-end applications, the type of attacks also vary.

In the rest of this section, we will explain two classes of vulnerabilities that are particularly relevant for our work.

2.1.1 Injection

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. In this case, we are dealing with a server-side attack. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

SQL injection (SQLi) is one of the most dangerous vulnerabilities that a Web App can be prone to. When a user's input is being passed unvalidated and unsanitized as part of an SQL query that means that the attacker can manipulate the query itself and force it to return different data than what it was supposed to return [8]. Figure 2.1 shows how to carry out an SQLi attack. The attacker inserts a SQL command or conditional logic into the input field, such as a student `ID` number of `117 OR 1=1;--` then, the resulting query is sent to the database (steps 1 and 2). Normally the query would search the student's table for the matching `ID` sent with the injected code. With the attacker, the predicate is always true because `ID OR 1=1;` executes to true independently of the value of `ID`. As a result, the database will return all data from the student's table back to the attacker (steps 3 and 4).

Another example of an SQLi is shown in Figure 2.2, which is a vulnerability that existed in previous versions of the tool *phpmyadmin*. Due to missing validation of the user-provided parameters (represented in Figure 2.2 as the variable `$scale`), it was possible to inject SQL code that would run with the privileges of the user control. This gives read and write access to the configuration tables of the database, and possibly read access to some other table for which the tool has the necessary privileges. The vulnerability was fixed by adding a numeric validation check to the `$scale` variable.

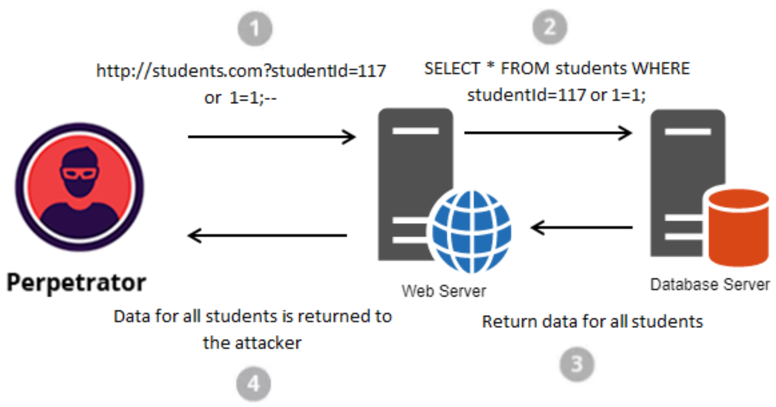

1
http://students.com?studentId=117 or 1=1;--
2
SELECT * FROM students WHERE studentId=117 or 1=1;
Perpetrator
Web Server
Database Server
Data for all students is returned to the attacker
Return data for all students
3
4

Figure 2.1: An example of the steps to perform an SQLi attack.

```php
$pmd_table = PMA_backquote($GLOBALS['cfgRelation']['db']) . '.' .
   PMA_backquote($GLOBALS['cfgRelation']['designer_coords']);
$pma_table = PMA_backquote($GLOBALS['cfgRelation']['db']) . '.' .
   PMA_backquote($cfgRelation['table_coords']);

if (isset($exp)) {

  $sql = "REPLACE INTO " . $pma_table . " (db_name, table_name,
   pdf_page_number, x, y) SELECT db_name, table_name, " . $pdf_page_number
    . ", ROUND(x/" . $scale . ") , ROUND(y/" . $scale . ") y FROM " .
   $pmd_table . " WHERE db_name = '" . $db . "'";

  PMA_query_as_controluser($sql,TRUE,PMA_DBI_QUERY_STORE);
}
```

Figure 2.2: SQLi vulnerability formed in *phpmyadmin*.

2.1.2 Cross-Site Scripting

Cross-Site Scripting, commonly abbreviated to XSS, refers to a client-side attack where an attacker uses a Web App to send malicious code, generally in the form of a JS script, to a different end-user. Since the browser of the user thinks the script came from a trusted source, the malicious script can access cookies, session tokens, or other sensitive information retained by the browser and used with that site [2]. Matthew et al. [29] refer that XSS vulnerabilities are difficult to prevent because it is tricky for Web Apps to anticipate the client-side semantics. An XSS attack occurs when:

- Data is inserted in a Web App through an untrusted source (most frequently a web request);
- Data is included in dynamic content that is sent to a user browser without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JS, but may also include HTML, Flash, or any other type of code that the browser may execute.

The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site. Since the malicious content runs with the same privilege as the trusted content from the web server, it can steal the victim's private data or take unauthorized actions on the user's behalf.

To prevent XSS vulnerabilities, all the untrusted content from users in web pages must be sanitized. However, proper sanitization is very challenging. One could let the server sanitize the untrusted content before delivering it to the browser. However, when a browser interprets certain content differently from what the server intends, attackers can take advantage of this discrepancy, as exemplified in the *Samyworm* [11], one of the fastest spreading browser worms to date. Alternatively, one could let the client sanitize untrusted content. However, without the server's help, the client cannot distinguish between trusted and untrusted content in the web pages.

There are three main categories of XSS attacks: *reflected*, *stored* and *DOM*.

With *reflected XSS*, the malicious script is not retrieved from storage but is instead reflected the user browser by the server. When a user is tricked into clicking on a malicious link, the injected code is included in the request that goes to the vulnerable web site, which sends the attack back to the user's browser. The browser will execute the script as the source is considered trusted.

As an example, in Figure 2.3, the perpetrator embeds a malicious script into a hyperlink, enabling the viewing of user session cookies. The link is sent to the victim via email to fool her to click on it. The script is executed by the Web App and reflected back to the victim's browser. Lastly, the browser sends the session cookies to the perpetrator, enabling access to the victim's private data.

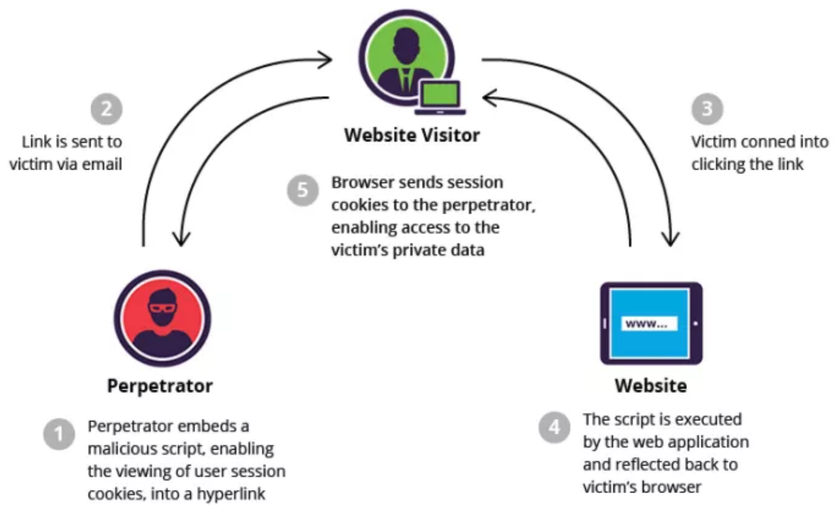

Website Visitor
2 Link is sent to victim via email
3 Victim conned into clicking the link
5 Browser sends session cookies to the perpetrator, enabling access to the victim's private data
Perpetrator
Website
www...
1 Perpetrator embeds a malicious script, enabling the viewing of user session cookies, into a hyperlink
4 The script is executed by the web application and reflected back to victim's browser

Figure 2.3: An example of the steps to perform *XSS reflected* [10].

XSS stored are based on the same principle of reflection of attacker-supplied data. The difference is that in these cases the vulnerable website stores the malicious script permanently and reflects/sends that data to any user accessing it. In this way, the same scripts, that are used to exploit XSS vulnerabilities by reflection can be used to exploit XSS vulnerabilities by storage. The victim retrieves the script from the server when it requests the stored information.

As an example, in Figure 2.4 we can see that the perpetrator discovers a website having a vulnerability that enables the script injection. The perpetrator injects a malicious script that steals each visitor's session cookies. This stealthy approach involves using JS to create a new, albeit broken, image that points to a cookie catching script. For each visit to the website, the malicious cookie catching script is activated. Finally, the visitor's session cookie is sent to the perpetrator.

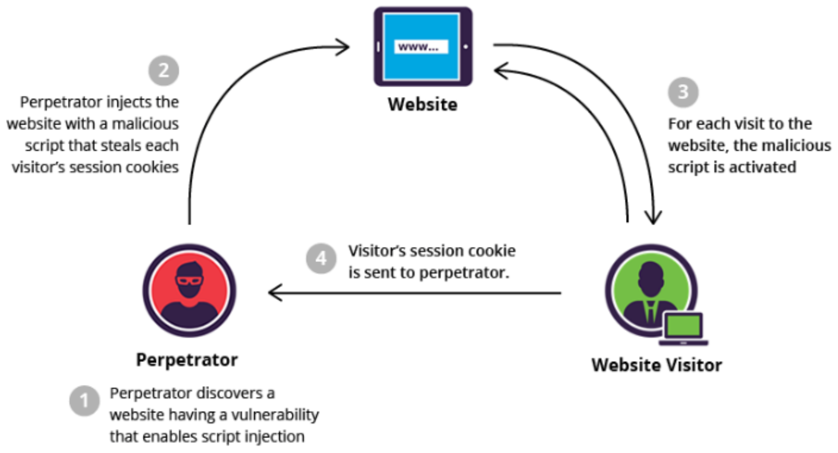

www...
Website
2
Perpetrator injects the website with a malicious script that steals each visitor's session cookies
3
For each visit to the website, the malicious script is activated
4
Visitor's session cookie is sent to perpetrator.
Perpetrator
Website Visitor
1
Perpetrator discovers a website having a vulnerability that enables script injection

Figure 2.4: An example of the steps to perform a *XSS stored* attack [10].

XSS DOM is an attack wherein the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser, so that the client-side code runs in an "unexpected" manner. That is, the page itself does not change, but the client-side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment. This differs from the stored and reflected XSS attacks, wherein the attack payload is placed in the response page due to a server-side flaw.

An example of code vulnerable to a XSS DOM is shown in Figure 2.5, based on a vulnerability that existed in previous versions of the *phpmyadmin* tool. The vulnerability was exploited by using specially crafted MySQL table comments that included a malicious script. An example of a malicious script would be an AJAX call that sends the contents of `document.cookie` to a website that the attacker controls. The attacker could then use the session cookie to log in as the victim. Later on, the development team fixed the vulnerability by applying the `htmlspecialchars()` sanitization function to line 2.

```
$browse_table_label = '<a href="sql.php?' . $tbl_url_query
    . '&amp;pos=0" title="' . $current_table['TABLE_COMMENT'] . '">'
    . $truename . '</a>';
```

Figure 2.5: A *DOM XSS* vulnerabilty in previous versions of the *phpmyadmin* tool.

2.2 Static Analysis

One way to deal with security vulnerabilities is to wait until the bugs are exploited by an attacker, then produce a patch that one hopes fixes the problem without introducing new flaws, and whine when system administrators do not install patches quickly enough [27]. Not surprisingly, this approach has proven largely ineffective.

The solutions for reducing software flaw damage can be grouped into two categories:

- Mitigate the damage that flaws can cause.
- Eliminate flaws before the software is deployed.

For the first category, there are techniques that limit security risks from software bugs, which include modifying program binaries to insert runtime checks or running applications in restricted environments that limit the harm they may do. The second category is related to techniques to detect and correct software flaws. They include human code reviews, testing, and static analysis. Functional testing is typically ineffective for finding security vulnerabilities.

Tosin et al. [45] states that developers do not code with the mindset of an attacker because they care more about delivering functionalities. Common coding mistakes and inadvertent programming errors are weaknesses that often evolve into exploitable vulnerabilities. The reality is that about 70-percent of reported attacks are performed at the application layer rather than the network layer [30]. Integrating Static Analysis Tools (SAT) could be envisaged to help developers program defensively.

Static analysis techniques analyse the source code directly. Thus, using static analysis lets us make claims about all possible program executions rather than just a particular test-case execution. From a security viewpoint, this is a significant advantage. There is a range of static analysis techniques, offering tradeoffs between the required effort and analysis complexity. At the low-effort end are standard compilers, which perform type the checking and other simple program analyses. At the other extreme are full program verifiers that attempt to prove complex properties about programs. They typically require a complete formal specification and use automated theorem provers. These techniques have been effective but are almost always too expensive and cumbersome to use on normal programs.

There are works that apply a lightweight form of analysis. Gustavo et al. [32] describe Splint, a tool that performs an analysis similar to those done by a compiler. Hence, they are efficient and scalable, and they can detect a wide range of implementation flaws by exploiting annotations added to the program the focus in not, however, looking for vulnerabilities.

Historically, static analysis tools were used to prove the absence of bugs inside a program, and they were particularly effective in specific application domains [32]. Typically, a predefined set of rules is used to find vulnerabilities, such as searching for the use of insecure library functions, buffer overflows or insufficient input data validation. One example of a basic static code analysis tool is *flawfinder* [5]. Flawfinder examines C and C++ source code and reports possible security weaknesses sorted by risk level. It does this by checking the code for potentially dangerous functions like `strcpy()`, which does an unbounded copy of a string from a source to a destination. The function does not check whether the destination buffer is big enough to store the source, which can easily lead to a buffer overflow if the programmer did not validate this beforehand.

Tools like *flawfinder* do not actually check for validation and leave it up to the user to verify. It is not hard to imagine that this leads to numerous false positives, which can be a little disheartening to the programmer who has to check each reported potential vulnerability manually. To provide better results, many tools employ a technique called taint checking as part of the static analysis. The main idea behind taint checking is that any variable that can be modified (either directly or indirectly) by input coming from an external user has to be considered tainted, as it has the potential to contain malicious data. Variables that are derived from tainted variables become tainted as well. Static code analyzers use taint checking by finding potentially vulnerable functions (so-called sinks) and then trace back their parameters to see if they were tainted. If a parameter is a constant variable set by the programmer, the analyzer will not report it. If the tool believes that the variable could be modified by the external user, it will list it as a potential vulnerability [40].

RIPS is used for the automated detection of security vulnerabilities in PHP applications [9]. It tokenizes the code (lexical analysis) based on a PHP tokenizer extension and performs semantic analysis to build a program model. It performs backward-directed inter-procedural taint analysis of sensitive sinks, based on previously analyzed variable assignments. Its strength is the ability to scan PHP applications very fast for PHP-specific vulnerabilities. It supports the detection of 15 different vulnerability types, including XSS, SQLi, Local File Inclusion (LFI), and others. However, experimental studies demonstrate that RIPS generates a high number of false positives because it does not use an abstract syntax tree or control flow graph, and lacks support for object-oriented code.

WAP is another static analysis and data mining tool for the detection and correction of input validation vulnerabilities in Web App written in PHP [13]. WAP detects and corrects several classes of vulnerabilities, including SQLi and XSS among others. This tool does taint analysis to track malicious inputs inserted at entry points (`$_GET, $_POST` arrays) and to verify if they reach some sensitive sink (PHP functions that can be exploited by malicious input). After the detection, the tool uses data mining to confirm if the vulnerabilities are real or false positives. In the end, the real vulnerabilities are corrected with the insertion of the fixes (small pieces of code) in the programs.

SATs play an important role to ensure the product meets the quality requirements. SATs exercise application source code and check for violations. The reality is that to date the vast majority of critical vulnerabilities are found by manual analysis of code by security experts. Despite this, different studies have investigated why developers do not use SAT to find bugs or how developers interact with such tools when diagnosing potential security vulnerabilities. Findings show that false positives and the way warnings are presented are barriers to use. Similarly, deep interaction by developers with the tool's results can create challenges of cognitively demanding tasks that could threaten the use of such tools [50]. Baca et al. [18] evaluated the use of a commercial static analysis tool to improve security in industrial settings. They found that, although the tool reported some relevant warnings, it was hard for developers to classify them. In addition, developers corrected false positive warnings, which created vulnerabilities in previously safe code. Hofer et al. [33] have used some other metrics to guide tools' selection such as installation, configuration, support, reports, errors found, and whether the tools can handle a whole project rather than parsing single files. Other researchers have also performed independent quantitative evaluations of static analysis tools with regard to their performance to detect security weaknesses [45].

2.3 Fuzzing

Fuzzing is one of the most effective approaches to find vulnerabilities in large software. The technique consists in feeding the target application with unexpected inputs to look for abnormal program termination. The crucial step in fuzzing is to choose relevant unexpected inputs, likely to reveal potential vulnerabilities [36].

Fuzzing was first introduced by Miller on a project to promote the reliability of Unix systems [36]. In this project, Miller confirmed that when arbitrary input values were delivered to the program under test, they created an exception and the program was shut down. This experience evolved into the fuzzing concept, which injects random values into the software with the expectation of bringing the execution to an expected state.

Fuzzing can be divided into *dumb fuzzing* and *smart fuzzing* depending on the way the inputs are generated. Dumb fuzzing is the simplest form of fuzzing technology because input values are produced with random data. This approach, however, has difficulty to find inputs that go beyond the initial validation layer of the software, and therefore code coverage is restricted.

Smart fuzzing generates appropriate values for the input format of the target through software analysis and error generation. However, there is a disadvantage in that it requires specialized knowledge to analyze the target software, and it may take a long time to generate a suitable model for the software. *Fuzzing mutation* is a test technique for modifying input data samples to make novel test cases that are tried in the target software [36]. Recently, an evolutionary technique has been introduced that generates a new input based on feedback on the response of the target software [36].

George et. al [39] examined 32 recently published works on fuzz testing and studied their experimental evaluations. They suggest a procedure that should be followed by any researcher that develops a new fuzzer algorithm (call it A). The researcher should empirically demonstrate that A provides an advantage over the status quo, namely:

- Choose a compelling baseline fuzzer B to compare against;
- Get a sample of target programs for the benchmark suite;
- Select a performance metric to measure A and B when they are run on the benchmark suite; ideally, this metric is the number of bugs identified by crashing inputs;
- Choose a meaningful set of configuration parameters, e.g., the seed file (or files) to start fuzzing with, and the duration of an experiment.

They found that none of the fuzz testing evaluations that were considered carried out all of the above steps properly (though some got close). This is bad news in theory, and after carrying out more than 50000 CPU hours of experiments, they believe it is bad news in practice too.

2.4 Symbolic Execution

Symbolic execution is a technique that explores feasible paths in the program by setting inputs to symbolic values rather than a real value [36]. It was first proposed in King's paper in 1975 [37]. This test technique was developed to verify if a specific condition in the software could be violated by the input values.

The technique can be divided into *offline symbolic execution* and *online symbolic execution*. Offline symbolic execution chooses only one path in the program to create a new input value by resolving the path predicate [31]. The program must be processed from the beginning to explore other paths, and so there are disadvantages because it causes overhead due to re-execution.

Online symbolic execution replicates states and computes novel path predicates at every point where the symbolic executor encounters a branch statement. There is no overhead associated with re-execution with the online method, but the downside is that it requires the storage of all status information and the simultaneous processing of multiple symbolic paths, leading to significant resource consumption.

In order to solve this problem, a hybrid form of symbolic execution was suggested. The *hybrid symbolic execution* saves state information as in online symbolic execution whenever a branch statement is executed and proceeds until memory is exhausted [22]. When there is no more space to save, there is a switch to the offline symbolic execution and a path search is performed.

In recent years, the *concolic execution* has been proposed, which is a method of testing by substituting an actual value (Concrete Value) in particular symbolic variable, and testing with a mixture of concrete and symbolic values. This technique is a technique of generates a new input value by solving a path expression branching statements. The reason for executing with that is constructed based on the actual value is that if the symbolic executor encounters a difficult problem (and it takes a long time or does not solve the problem) then the test could no longer be performed. However, if the actual value is substituted, a deeper path search becomes possible.

Abeer et al. [17] pursue an automated exploit generation approach in Web App. Given an application, the goal is to automatically construct a sequence of malicious HTTP request inputs that direct an application's execution to a vulnerable sink. The starting point to the approach is static analysis: the creation of models of the Web App behavior along its paths that are based on symbolic execution. From here, there are two scalability challenges that must be overcome to make the exploit finding successful. The first one involves path selection: what are the paths that one must explore to make opportunistic exploit generation successful? The approach makes the observation that it is possible to prioritize the traversal of the paths by using their constraint solving costs, such that one we efficiently identifies paths that lead to a successful exploit. The second issue is about persistent database states: how to deal with database queries that may be present along the paths that are explored. This issue becomes particularly important in the context of second-order attack creation, where a vulnerable query, say, is exploited to store some data that is subsequently read from a (second) exploit sink.

Cristian et al. [20] presented KLEE, a symbolic execution tool capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. When KLEE runs a program, it tries to explore every possible path. This is done by executing the program symbolically, i.e., tracking all constraints on inputs marked symbolic as each instruction is run. When a conditional that depends on a symbolic input is encountered, a constraint solver is used to determine which direction the path will follow. In some cases execution is not constrained to follow a single path, the condition can be true or false depending on the input, and the execution conceptually forks. When this happens, KLEE clones the current process and follows both paths, adding the appropriate constraint to the path conditions of each process.

Torben et al. [34] present THAPS a vulnerability scanner for Web Apps written in PHP. THAPS is fundamentally a taint analysis tool based on the symbolic execution of PHP. It is in the symbolic execution engine that vulnerabilities are detected and where reports are generated, containing details of each (potential) bug. To identify vulnerabilities, the taint analysis identifies where user input is able to enter the application, called a source, and how it is propagated through the application. If the tainted data reaches critical points of the application, where it is able to alter the outcome of the application, it has reached a *sink*. Every time tainted data reaches a sink without being properly sanitized first, a vulnerability is reported. To simulate all the possible outcomes the analysis might need to store several (many) values for the same variable because of assignments inside different code branches. Whenever there are multiple values a simulation has to be performed on each of these. THAPS stores the values in a variable storage, which also records what branch of the code the value belongs to.

2.5 Oracles

A test oracle, or just oracle, is a mechanism for determining whether a test has passed or failed. One of the most effective ways to produce test oracles is to use a model of the target software and generate complete tests, including both input data and expected results, directly from the model. The model, in this case, is exactly what the name implies: it incorporates the most important aspects of the target, but not every detail (if it did include all details, it would be equivalent to the system itself) [7].

The most common form of oracles is proxy servers. A proxy can be placed between a service and a client in order to monitor what kind of traffic is going through. Divya et al. [44] present *FlowWatcher*, an HTTP proxy that mitigates data disclosure vulnerabilities in unmodified Web Apps. FlowWatcher monitors HTTP traffic and shadows part of an application's access control state based on a rule-based specification of the *user-data-access* (UDA) policy. The UDA policy states the intended data ownership and how it changes based on observed HTTP requests. FlowWatcher detects violations of the UDA policy by tracking data items that are likely to be unique across HTTP requests and responses of different users.

Anyi et al. [41] propose an *SQL Proxy-based Blocker* (SQLProb). SQLProb harnesses the effectiveness and adaptivity of genetic algorithms to dynamically detect and extract users' inputs for undesirable SQL control sequences. Compared to state-of-the-art protection mechanisms, this method does not require any code changes on either the client, the web-server or the back-end database. Rather, the system uses a proxy that seamlessly integrates with existing operational environments offering protection to front-end web servers and back-end databases.

Iberia et al. [42] proposed a way to detect and block attacks at runtime without programmer intervention inside a database. They call this approach *SElf-Protecting daTabases preventIng attacks* (SEPTIC). The paper focus is on the two main categories of attacks related to databases: SQLi attacks, which continue to be among those with the highest risk and for which new variants continue to appear, and stored injection attacks, which also involve SQL queries. For SQLi, they propose detecting attacks essentially by comparing queries with query models that were previously learned in an earlier training phase. For stored injection, they propose having plugins to deal with specific attacks before data is inserted in the database.

Chapter 3

Vulnerabilities Detection at Run Time and Continuous Auditing

This chapter focusses on the detection of Cross-Origin Resource Sharing (CORS) vulnerabilities. It proposes an architecture for a new vulnerability detection tool along with the key concepts and modules it relies on. The architecture resorts to crawling techniques to find links (external and internal) in the Web App and analyses them to check for possible dangerous strings related to CORS. The analysis is targeted to web pages produced/returned by the Web App and is performed at run time, allowing for Continuous Auditing (CA). The chapter is organized as follows: Section 3.1 provides an overview of CA. Section 3.2 describes the implications of CORS in modern Web Apps and the problems they may bring. Section 3.3 offers, respectively, an overview of the architecture, and a more detailed look into each module that composes the architecture.

3.1 Continuous Auditing for Detecting Vulnerabilities

Organizations are subject to various types of audits for different purposes, such as quality, environmental, operational or management. These processes follow a well defined time schedule to validate and analyse what has already been done and the current state of the organization. However, the competitive demands for organizations and, consequently, the need for innovation, promoted changes in the way web audit is done. CA assists the auditor in developing his or her opinion since it enables the evaluation of the relevant events that are observed in real-time, using automated and continuous processes. According to Camargo [21], CA is related to:

produces results [. . .] within a short period of time after the occurrence of a relevant event, i.e., it performs process control tests continuous] [. . .] [using] technology tools, [. . .] [allowing] identify nonconformities, trends, and risk indicators

Silva [52] observed that the execution of CA is based on the date surrounding a given event, and whenever possible in real-time. The data supporting the analysis should be reliable, assisting auditors and managers in decision-making and also facilitating early detection of fraudulent reports.

According to Costa [24], to conceive a CA comprehensive approach, there is the need to achieve resource optimization and rationalization through continuous and integrated types of audit, through which the same entity is subject. Here, CA contributes not only to the traditional monitoring tasks but also to the efficiency of an entity. Table 3.1 shows the fundamental differences between a Traditional Audit (TA) and a CA.

Traditional Auditing	**Continuous Auditing**
Held periodically	Continuously performed
Reactive approach	Proactive approach
Manual process	Automated process
Sampling tests	Whole population based tests
Tests take performance and human judgment into account	Tests take modeling of analytical data for subsequent monitoring into account
Reports are prepared periodically	Reports are produced continuously or frequently

Table 3.1: Traditional Auditing vs Continuous Auditing.

In the context of security, the concept of CA emerges as a novel audit paradigm likely to respond to new needs of a very dynamic environment where novel software is continuously developed and integrated into production systems. The traditional model of auditing, based on punctual and discontinuous analyses, becomes increasingly inadequate to the current dynamics of information and the systems that manage it. The application of updates and changes to the system configurations can introduce vulnerabilities and leave an organization susceptible to attacks. Therefore, to keep the data secure, one should check systems and devices continuously to detect flaws as they are discovered and reported to the organizations [12].

Pinto et al. [46] mention that there is a change in the traditional philosophy of "look back" audit to a CA paradigm that makes it possible to take corrective actions earlier. As such, there is a need for organizations to adopt a methodology that allows independent auditors to track the occurrence of events over the life of the system. These events, when monitored in real-time, allow audit parameter deviations to be found and reported, allowing to increase the speed and effectiveness of the responses by the decision-makers.

Continuous security auditing is intended to ensure real-time monitoring of the systems and the risks to the company's assets, not only to assess the current security level as well as monitor the system in real-time, increasing the efficiency of vulnerability discovery and mitigation. The information monitoring and review influences the risk approach both in terms of methodology and tools based on various tests.

Intrusion tests, for instance, help to identify weaknesses and potential improvements to complement the continuous audit, as they allow the identification of vulnerabilities and with this information simulate the behavior of an attacker. Since it is a highly specialized and technical type of test, it delivers immense value to organizations with the reported information. However, they might be limited in target, time, and radius and sometimes they depend on legal issues. For a continuous process where there is no such intrusive behavior, vulnerability analysis can be carried out with the aid of automatic tools over time to observe system status and take corrective actions to fix identified problems. In our work, we follow this last approach, to continuous monitor Web App while searching for the introduction of new CORS vulnerabilities.

3.2 CORS Exploitation

Web Apps play an important role in most organizations. CORS is a mechanism that uses additional HTTP headers to tell a browser to let a Web App running at one origin (domain) to have permission to access selected resources from a web server at a different domain. CORS breaks the limitations imposed by the Same-Origin Policy (SOP) of traditional HTML, allowing exchanges between various web servers in different networks. A Web App executes a cross-origin HTTP request when it asks for a resource that has a different origin (domain, protocol, and port) than its own origin. CORS is mainly implemented with the `XMLHttpRequest` (XHR), which is supported nowadays by almost every browser.

According to Chou [23], a webpage can retrieve data from a URL via XHR without refreshing its page. XHR also plays a very important role in AJAX applications. With AJAX, the JS creates XHR objects that can make requests and receive responses asynchronously, updating the display as responses are received. For example, consider Google Maps. When someone goes to Google Maps and searches for a location and slides the map, the browser uses AJAX to make calls to the server to retrieve the images that make the map. In this way, the map changes dynamically, but the remainder of the page is static and does not need to be redrawn with each alteration.

Jorg et al. [49] referred that XHR allows a web page to send arbitrary HTTP requests to any web server. This is different from just opening an URL or submitting an HTML form since with XHR the web page has full control over all HTTP headers to address the sort of issue, the CORS standard was developed to enable controlled cross-domain requests to be done:

- In a preflight request, the browser asks for access to a resource in a different domain. It sends an origin header with the source domain (*Origin: http://a.com*) to the target web server requesting CORS privileges;
- The target web server may answer with an error message (access denied) or with a CORS header (such as *Access-Control-Allow-Origin: http://a.com*) to grant the access;

- Based on the response, the browser proceeds with the execution.

The XHR object has a number of methods and properties. The following are the most used to perform a Cross-Origin request:

- *open*: Specifies the properties of the request, but does not actually initiate a connection.
- *send*: Creates the connection to this property and specifies the function that should be called when the state changes to ready.
- *setRequestHeader*: Defines the header value of an HTTP request. It is called after calling the `open()` method, but before invoking the `send()` method. If this method is called many times with the same header, the values are added to form a single HTTP request header.
- *getAllResponseHeaders*: Returns all the response headers, separated by Carriage Return Line Feed (CRLF), as a string, or returns null if no response has been received yet. If a network error happens then an empty string is returned.
- *readyState*: The ready state is a property of the XHR object. It enables the scripting code to determine in what state the response from a server is. It has five possible values:
  - 0: The request is uninitialized;
  - 1: The request has been set up;
  - 2: The request has been sent;
  - 3: Waiting for response;
  - 4: The response has arrived and is complete.

As mentioned before, under the restrictions imposed by the SOP, XHR can only be used to access a URL in the same domain. The restriction was eased when CORS appeared. CORS allows developers to send cross-domain requests using similar codes for requests inside the same domain. The concern about using CORS is that it increases the chances of cross-site attacks. A hacker can easily use the features of CORS to launch Cross-Site Request Forgery (CSRF) attacks. Farah et al. [28] describe a CSRF attack as follows: First, the victim user logs into the target website and gains access. The hacker then tricks the victim user, who visits a malicious website, into unknowingly running a malicious script that sents requests through the browser to the target website. Since the hacker script uses the victim's privileges to transmit the request, it can access the confidential data of the victim in the target site. This attack is difficult to identify in the system log as there are no unusual events.

Three common vulnerabilities found on CORS are: misconfigured wildcard (*) in the CORS headers; Trusting pre-domain wildcard as origin; and using XSS to make requests to cross-origin sites.

The first vulnerability is related to misconfigurations and incorrect usage of wildcards such as (*), under which domains are allowed to request resources. This is usually set as default, which means that any domain can access resources on the target site. As an example, we can consider the request below that is performed by the browser to the target:

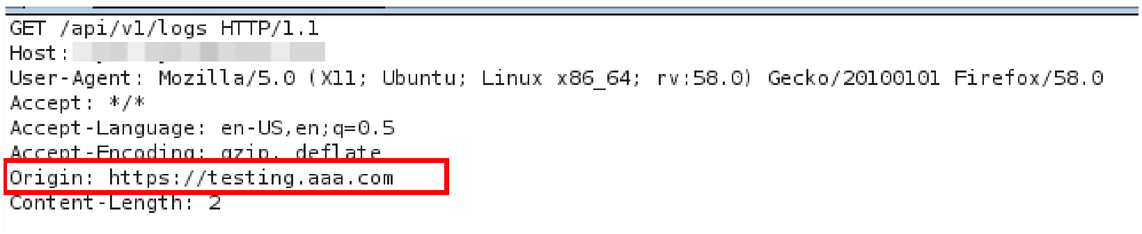
```
GET/api/userinfo.php
Host: www.victim.com
Origin: www.victim.com
```

When the above request is sent by the browser, the following response might be received:

```
HTTP/1.0 200 OK
Access-Control-Allow-Origin:*
Access-Control-Allow-Credentials:true
```

In this example, the Access-Control-Allow-Origin header is configured with a wildcard (*). It means that any domain can access the resources in the target web site.

In Figure 3.1, we can see an example of the exploitation of this sort of vulnerability, where the attacker can fetch user information like Name, User-ID, and Email-ID and send this information to an external server. To achieve this, he modifies the REQUEST Origin to the attacker domain.



```
GET /api/v1/logs HTTP/1.1
Host: [redacted]
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Origin: https://testing.aaa.com
Content-Length: 2
```

Figure 3.1: Tempered origin URL under REQUEST [1].

Since the target site shares information with any site, it can be exploited by using a domain *https://testing.aaa.com* (see Figure 3.1). The malicious server of domain *https://testing.aaa.com* embeds in the returned pages to the browsers some exploit code to steal confidential information from the vulnerable application (in *www.target.com*). When users open *https://testing.aaa.com* in the browser, the browser retrieves the sensitive information and sends it to the attacker server.

The second form of CORS is when a misconfiguration allows information sharing with domain names that are partly validated. For example, consider the following request:

```
GET/api/userinfo.php
Host: provider.com
Origin: requester.com
```

And the response to the above request would be:

```
HTTP/1.0 200 OK
Access-Control-Allow-Origin:requester.com
Access-Control-Allow-Credentials:true
```

Now, imagine that the user browser makes the following request, indicating as Origin a malicious web server:

```
GET/api/userinfo.php
Host: example.com
Connection: close
Origin: attackerrequester.com
```

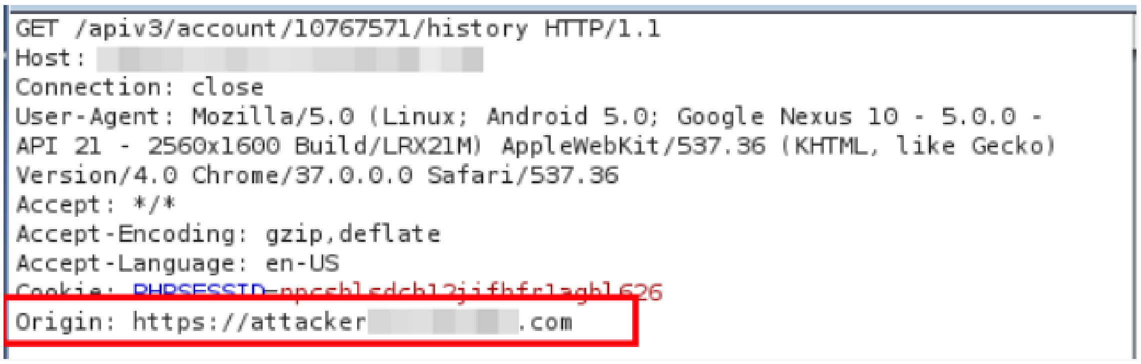
The unassuming target server would respond with the following answer, allowing requests from an attacker controlled server:

```
HTTP/1.0 200 OK
Access-Control-Allow-Origin: attackerrequester.com
Access-Control-Allow-Credentials: true
```

The reason why this permission might be given is a possible backend badly configured validation such as the following, which allows any origin to make requests as long as the domain ends with *requester.com*:

```
if ($_SERVER['HTTP_HOST'] == '*requester.com')
{
//Access data
else{ // unauthorized access}
}
```

In Figure 3.2, the host domain *provider.com* trusted all origins that ended with host name *requester.com* such as *attackerrequester.com*. So, the attacker tempered the origin header to *attackerrequester.com* and proceeded with the request. This can be exploited



```
GET /apiv3/account/10767571/history HTTP/1.1
Host: [redacted]
Connection: close
User-Agent: Mozilla/5.0 (Linux; Android 5.0; Google Nexus 10 - 5.0.0 -
API 21 - 2560x1600 Build/LRX21M) AppleWebKit/537.36 (KHTML, like Gecko)
Version/4.0 Chrome/37.0.0.0 Safari/537.36
Accept: */*
Accept-Encoding: gzip,deflate
Accept-Language: en-US
Cookie: PHPSESSID=npcshlsdch12jifhfr1aghl626
Origin: https://attacker[redacted].com
```

Figure 3.2: Response from the REQUEST [1].

the same way as in the first misconfiguration. The attacker can create a new domain with the name consisting of the whitelisted domain name. Then, he can embed that malicious site with exploits that will fetch sensitive information from the victim's site.

The third form of attack is related to the fact that a usual defense mechanism that developers employ against CORS exploitation is to whitelist domains that frequently request access for information. However, this solution might not be entirely secure because even if one of the subdomains of the whitelisted domain is vulnerable to other exploits such as XSS, it can enable CORS attacks to succeed. The following code shows the configuration in the victim web server that allows subdomains of *requester.com* to access resources of *provider.com*:

```
if ($_SERVER['HTTP_HOST'] == '*.requester.com')
{
//Access data
else{ // unauthorized access}
}
```

Lets imagine that users have access to *sub.requester.com* but not *requester.com*, and lets assume that *sub.requester.com* is vulnerable to XSS.

The attacker can exploit *provider.com* by using a XSS attack. Two applications exist in different domains. The provider CORS application is hosted on *provider.com* and another application is hosted on *pavan.requester.com* which is vulnerable to XSS.

Using this vulnerable XSS subdomain, we are able to fetch sensitive information from *provider.com*. The attacker injects a malicious JS payload in the "Name" parameter of a request to *pavan.requester.com*. When the page loads, the script gets executed and fetches sensitive information from the *provider.com*.

3.2.1 Summary of SOP and CORS

Many developers misunderstand the SOP and what CORS brings to the table. There are many badly informed developers stating that SOP prevents cross-site requests, and therefore avoids CSRF. This is not the case! All that SOP does is prevent a response from being read by another domain (origin). This is irrelevant to whether a CSRF attack is successful or not. The only time SOP comes into play with CSRF is to prevent any token from being read by a different domain.

As a conclusion, what CORS does is relax SOP. It does not increase security, as it simply allows some exceptions to take place. Some browsers with partial CORS support allow cross-site requests. However, they do not allow custom headers to be appended. As we saw in the examples, in CORS supported browsers, the *Origin* header cannot be set, preventing an attacker from spoofing this information.

Given the above, our solution to detect CORS is not related to additional HTTP headers that CORS uses but instead, our approach crawls the JS code to find interesting strings related to CORS. As mentioned, CORS uses the JS XHR object to make requests for data and providing more interactivity to the web page. The intention is to find functions like *open*, *send*, *setRequestHeader*, *getAllResponseHeaders* or *readyState* which represents that the Web App is using CORS.

3.3 Architecture to Support CORS Detection

This section presents the architecture of our solution along with the key concepts and modules it relies on. The architecture enables the analysis of web pages produced/returned by a Web App with regards to possible vulnerabilities related to external and internal links, which are associated with CORS. For the architecture presented, the term external link is associated with the HTML `href` attribute of a given external web page URL, which specifies the linked resource. Hyperlinks require the `href` attribute because it specifies a location, i.e., the URL of the page where the link goes to. In the following example, the link News has a hyperlink to the external resource `https://provider.com/news/`.

```
<a href="https://provider.com/news/">News</a>
```

An internal link is associated to a resource on the same domain. The attribute `src` in a tag is the path to a file or resource that the HTML document wants to reference. For example, if we had a custom JS file named `script.js` and wanted to add its functionality to our HTML page, we would point to the file `script.js` in the *.html* file:

```
<script src="../news/js/script.js"></script>
```

For the internal links, as mentioned before, the approach also uses crawling techniques in the web pages produced/returned by a Web App to look for interesting strings related to CORS. In particular, it determines if the source code associated with the internal link, usually a JS or AJAX, contains functions related to CORS, such as the ones presented in the previous section.

The main components of the architecture are presented in Figure 3.3. It is composed of three main modules as follows:

1. *Web Crawler*: this is an engine that navigates through the HTML DOM (Document Object Model) tree of the web pages, to get nodes that might be of interest to be further analysed and monitored. Sometimes it is called a spider or spiderbot and is used for data collection in many application areas. It is able to search, extract and collect links, data, images, and emails from web sites;

2. *Storage*: this module is responsible for storing the results from the web crawler module for future use. All data searched and collected by the web crawler is stored in a database. It supports the same CA engine to store the audit results and the current state of web pages produced/returned by the Web App. Storage employs the traditional relation mode that is a representative data model, suitable as the underlying model of a Relational Database Management System (RDBMS), which is based on the principle that all data is stored in tables;

3. *Continuous Auditing*: this module implements the mechanism that allows CA of the monitored sites. It is composed of two submodules: *Reputation & Assessment* and *Site Internal Analysis*. The submodules are responsible, respectively, for analysing

periodically the external and internal links, and determining the current state of the web pages produced/returned by the Web App. The first submodule uses a Virus Total (VT) Application Programming Interface (API) to analyse the external URLs found. VT is an online tool that allows analysing files and URLs, enabling the identification of malicious content detectable by antivirus and website scanners. The second submodule is responsible for detecting CORS callings in the internal links identified for the Web App. For the detection, it searches for strings related to CORS in the Web App JS pages.

Each of the modules and submodules has a specific task in the application.

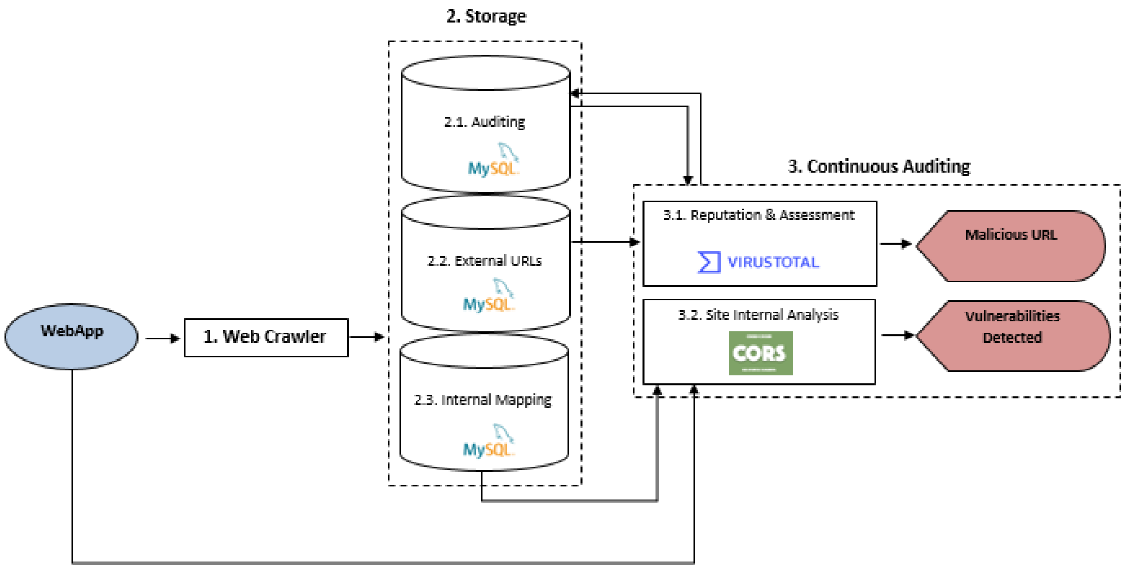


Figure 3.3: Vulnerabilities Detection at Runtime and CA architecture.

3.4 Main Modules

This section provides a more detailed description of the components that form the architecture.

3.4.1 Web Crawler

The *Web Crawler* is a program that visits web sites, and obtains their pages plus other data in order to create indexes, working as a librarian. The crawler looks for information on the web sites, which it assigns to certain categories. Then, the pages are indexed and cataloged so that the obtained information is retrievable and can be evaluated. The crawler works as a service that is programmed to visit sites that have been submitted by their owners for testing, as new sites or sites that have been updated.

Our crawler is capable of navigating through the DOM tree of the web pages. According to the DOM specification, every HTML tag is an object. Nested tags are "children" of the enclosing one. All these objects are accessible using JS and we use them to find the links in the `a href` tag, for the external links, and `script src` tag for the internal links.

3.4.2 Storage

The *Storage* module employs a database using a relational model. This module stores the findings collected during the crawling search, i.e., all the external and internal links found during the analysis of the web pages. This information is stored to facilitate future analysis since the site is crawled more than once, we associate to each interaction of information collection a version number. Figure 3.4 shows the database model including the following tables:

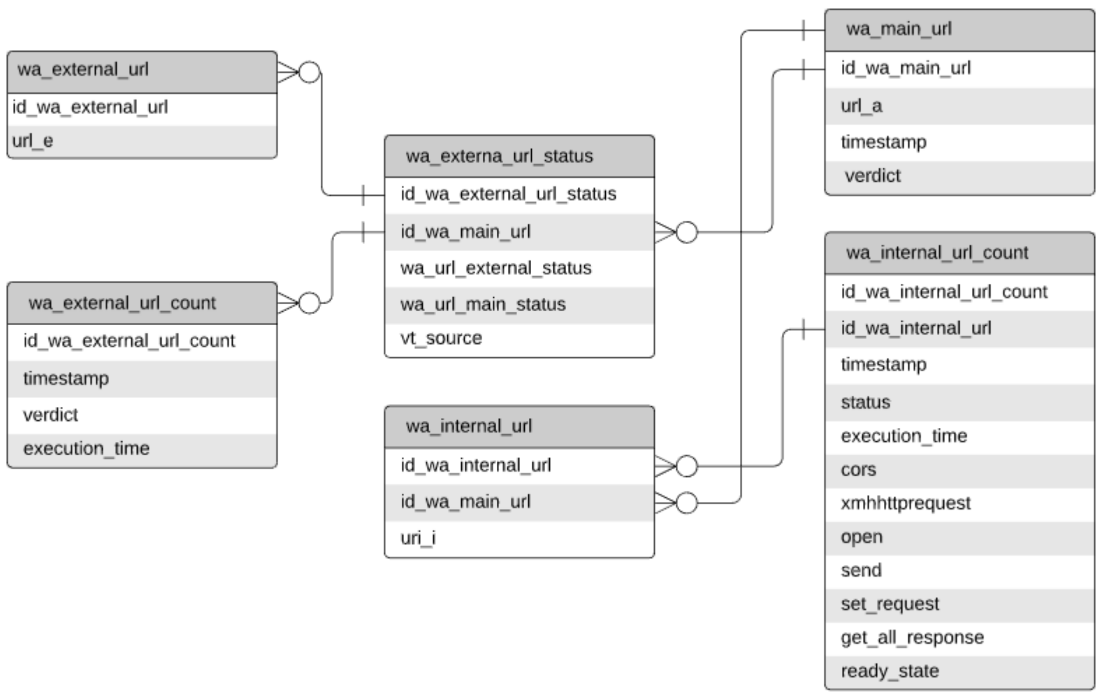


Figure 3.4: Storage database model.

- *wa main url*: Stores the URL of the web pages of the Web App that will be monitored. It uses a field to store the URL to analyse, another to log the time when the request was made and one more to save the verdict returned by the Virus Total (VT) analysis to determine the Web App state.
- *wa external url*: Keeps all the external URLs that were found during the crawling phase. It saves the external URLs of the analysed Web App.

- *wa external url status*: Stores the sources of the VT analysis and the status of the analysis. It has fields to save the date of the analysis, the source of the analysis and the status of the URL as *benign* or *malign*.
- *wa external url count*: Maintains the information related to indicators like execution times and the number of URLs.
- *wa internal url*: Stores all the internal links found during the crawling phase of the web pages, and that were returned by the Web App.
- *wa internal url count*: Saves the information related to CORS on the internal links found during the crawling phase of the CA module and indicators like execution times, number of URLs and XHR methods called.

3.4.3 Continuous Auditing

The *Continuous Auditing* module ensures real-time monitoring of the Web App. This module works after a Web App is first analysed. It uses the Operating System (OS) Task Scheduler to periodically perform a routine task, which supports the analysis of the URLs of the WA already stored. It is composed of two submodules called *Reputation & Assessment* and *Site Internal Analysis*.

The CA module collects the URLs from the Web App stored in the database and already analysed. Then, it will crawl the Web App pages to verify the prior assessment of external links by resending the links found during the new crawling to the VT using the *Reputation and Assesment* submodule.

The *Site Internal Analysis* submodule will search for internal links and re-analyse them to detect CORS callings. It will search for interesting functions in the web pages produced by the Web App. The submodule performs a static analysis of the code. It searches for the JS `script src` tag to get the URLs and carry out a crawling to web pages returned by the Web App to find if there are new interesting strings related to CORS calls. It searches all the interesting strings related to JS XHR objects, which have the methods `open`, `send`, `setRequestHeader`, `getAllResponseHeaders` or `readyState` are `quered`. If these strings are found then it is possible that the Web App has a CORS vulnerability.

In both submodules, new links may be found, either due to changes made by the developers or by an attacker.

Chapter 4

Implementation of VuDRuCA

This chapter presents our current implementation of the proposed architecture, in a tool we call VuDRuCA. Section 4.1 describes the main components of the tool and explains some aspects of the code we developed, and Section 4.2 gives an example of the tool execution and CORS vulnerability detection.

VuDRuCA is a tool for CA in Web Apps regarding its external and internal links. As a web server PHP interpreter solution to store and run the Web App, we used a free and open-source cross-platform stack package developed by Apache Friends named XAMPP (Windows, Apache, MySQL, PHP and Perl). It consists of an Apache HTTP Server that interprets scripts written in PHP and Perl programming languages. Since most actual web server deployments use the same components as XAMPP, it makes transitioning from a local test server to a live server possible. Since XAMPP supports PHP, we used this open source general-purpose scripting language to build our tool, considering that it is especially suited for web development and can be embedded into HTML.

4.1 Main Modules

As a backend system storage model, the tool uses a MySQL Relational Database Management System (RDBMS). Next, we will describe the modules that compose it:

- *Web Crawler module*: this module uses the Virus Total (VT) API and the PHP language with the Client URL Library (CURL) to navigate on HTML DOM tree of the Web Apps. The crawler searches for external and internal links. Therefore, each page (HTML and PHP) of the Web App is visited by the crawler, looking for internal and external links to store them in the database, through the storage module. CURL provides a library (libcurl) for transferring data using various protocols, including HTTP and HTTPS, which are the ones we need to focus on.

- *Storage module*: this module uses a MySQL RDBMS to create, update, administer and interact with a relational database. All the external and internal links, the results of the VT analysis and the detection of CORS methods are stored on database tables with the associated classification (malign/benign, CORS detected/not detected).

- *Continuous Monitoring module*: this module is responsible for the CA monitoring of the analysed Web Apps. All the URL links of the Web App stored in the database are crawled once after the first analysis takes place. The process to find external and internal links will be repeated periodically. To do this, the OS task scheduler is used to call a routine every 24h. This way, for the links that already exist in the database, the module executes two tasks. First, it requests to VT the current state of the external links. To do so, VT receives a link and performs an inspection over 70 antivírus scanners and URL/domain blacklisting services. Second, it analyses the JS files associated with the internal links in order to find CORS vulnerabilities.
- *Console to display warnings*: For the external links the tool will give a verdict based on the VT result with color:
  - *Green*: The external link was analysed by VT and is not malicious.
  - *Red*: The external link was analysed by VT and is classified as malicious by antivirus sources.
  - *Black*: The tool did not find external links in the Web App.

  For the internal links and CORS findings, the tool will give a verdict with color:
  - *Green*: The internal links were checked and there are no matches for CORS.
  - *Red*: The internal links were checked and there are matches for CORS.
  - *Black*: The analysis did not find internal links in the Web App.

For the external links, it uploads, scans and returns a report without the need of using the VT website interface. PHP scripts interact with the VT public interface and access the information generated by it. In order to use the API, we sign up with the VT Community and get a public API key that is used in all scripts. The scripts resort to the CURL library to transfer data from the Web App. For the internal links, we use a method that searches for interesting strings related to CORS.

Basically, VuDRuCA has two principal routines: Main engine and CA. Figure 4.1 illustrates the former and Figures 4.2, 4.3, 4.4 represent the later.

In the next two sections, we will describe the routines that were used in the VuDRuCA main engine and the 3 routines that constitute the CA engine.

4.2 Main Engine Routines

The main engine carries out the analysis of the Web Apps submitted to VuDRuCA. Next, we will describe the five routines that compose the engine according to Figure 4.1.

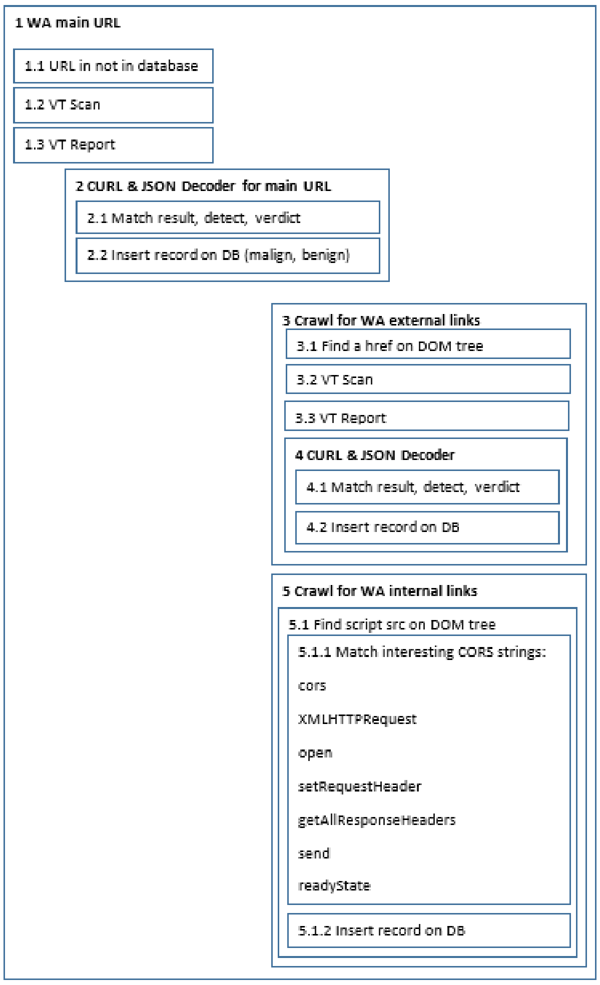

1 WA main URL
1.1 URL in not in database
1.2 VT Scan
1.3 VT Report
2 CURL & JSON Decoder for main URL
2.1 Match result, detect, verdict
2.2 Insert record on DB (malign, benign)
3 Crawl for WA external links
3.1 Find a href on DOM tree
3.2 VT Scan
3.3 VT Report
4 CURL & JSON Decoder
4.1 Match result, detect, verdict
4.2 Insert record on DB
5 Crawl for WA internal links
5.1 Find script src on DOM tree
5.1.1 Match interesting CORS strings:
cors
XMLHTTPRequest
open
setRequestHeader
getAllResponseHeaders
send
readyState
5.1.2 Insert record on DB

Figure 4.1: VuDRuCa modules routines.

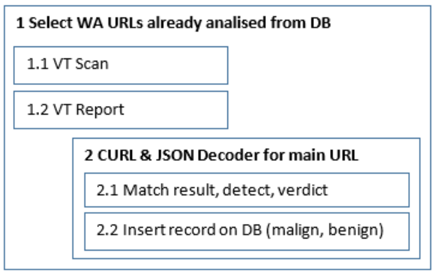

1 Select WA URLs already analised from DB
1.1 VT Scan
1.2 VT Report
2 CURL & JSON Decoder for main URL
2.1 Match result, detect, verdict
2.2 Insert record on DB (malign, benign)

Figure 4.2: VuDRuCA CA for main Web App URL routine.

- *Web App main URL*: This routine is responsible for the analysis of the main URL of the Web App. The main URL is sent to VT to be checked. To analyse the URL, VT will perform a scan of the site associated with the URL and then it will return a report describing the main findings.

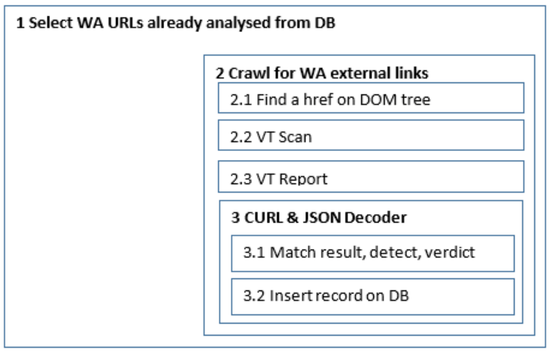

1 Select WA URLs already analysed from DB
2 Crawl for WA external links
2.1 Find a href on DOM tree
2.2 VT Scan
2.3 VT Report
3 CURL & JSON Decoder
3.1 Match result, detect, verdict
3.2 Insert record on DB

Figure 4.3: VuDRuCA CA for external Web App links routine.

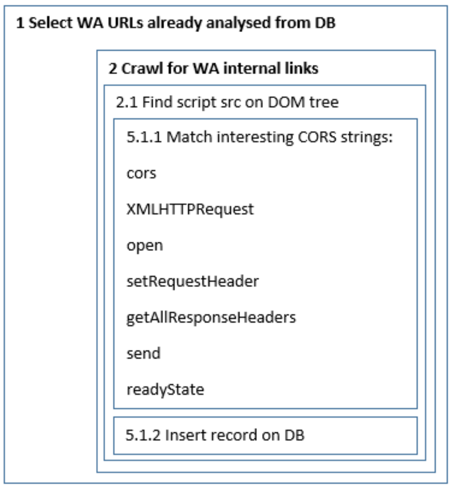

1 Select WA URLs already analysed from DB
2 Crawl for WA internal links
2.1 Find script src on DOM tree
5.1.1 Match interesting CORS strings:
cors
XMLHTTPRequest
open
setRequestHeader
getAllResponseHeaders
send
readyState
5.1.2 Insert record on DB

Figure 4.4: VuDRuCA CA for internal Web App links routine.

- *CURL & JSON Decode for main URL*: The VT report is returned in a JSON response. It needs to be parsed to get the values to be served in an associative array. For our study, the keys that are required are the result, detect and verdict. The information is then stored in the database.

- *Crawl for Web App external links*: After analysing the main URL, this routine crawls the files of the Web App to find external links. To do this, it parses the DOM tree of the analysed URL to find the tag attribute `href`. The found URLs are classified as external if they have http:// or https://. The external links will be sent to VT to perform a scan and get a report as for the main URL Web App.

- *CURL & JSON Decoder*: The returned JSON response is also parsed to obtain the verdict for the external links as benign or malign.

- *Crawl for Web App internal links*: If a URL is classified as internal then the corresponding DOM tree is searched to find URLs on `script` tags. All the internal URLs are parsed to find strings related to CORS. A few examples are: `cors`, `XMLHttpRequest`, `open`, `setRequestHeader`, `getAllResponseHeaders`, `send` and `readyState`. The results are also stored in the database.

4.3 Continuous Auditing Engine Routines

The CA engine consists of three routines. One routine serves to crawl the Web App's main URLs already entered in the database (Figure 4.2). The other two routines allow the CA engine to check the external and internal links of the Web Apps (Figures 4.3 and 4.4). The method used to crawl is similar to the one used in the first analysis (described in the previous section). It uses CURL and the VT API for the external links and interesting CORS strings for the internal links.

- *Continuous Auditing routine for main Web App URLs*: The routine for the main URL analysis uses the same approach as the first analysis. First, it gets the Web Apps URLs from the database and performs a scan, followed by producing a report using the VT API. After it parses the returned JSON response to get the values for the keys result, detected and verdict.

- *Continuous Auditing routine for external URLs*: This routine gets all the main URLs from the database. As performed by the main engine routine, it uses a crawler to get the links passed on the `href` attribute and sends them to be analysed by the VT API. The external links found are saved to compare with the initial analysis and find if there are new external links in the Web Apps.

- *Continuous Auditing routine for internal URLs*: This routine obtains all the URLs from the database and uses the crawler to get the links passed on the `src` attribute. It then verifies if these links are performing CORS calls or if there are more new internal links that do a CORS call.

```php
$sql5 = "SELECT * FROM db_site.webapp_to_analyse WHERE url_a='$url_a';";

$result = $conn->query($sql5);

if ($result->num_rows != 0)
{
    $message=($url_a . " is already analysed and in continuous auditing!!!!");
    echo "<script type='text/javascript'>alert('$message');</script>";
}
```

Listing 4.1: Verifying if the Web App is already analysed and in continuous auditing.

Next, we will present some examples of the PHP code that implements the described routines. The first subroutine determines if the main URL of the Web App is already in the database since it is a core task of the solution is the CA. If the Web App URL is already in the database, then a CA is in course. A subroutine using a SQL query will get all the Web App URLs already stored and analysed to perform this check (Listing 4.1).

If the URL for the Web App to analyse is not in the database, a subroutine will check if the request follows the HTTP format, using a regular expression (Listing 4.2).

```php
if(!preg_match('/\b(?:(?:https?):\/\/|www\.)
    [-A-Z0-9+&@#\/%=~_|$?!:,.]*[A-Z0-9+&@#\/%=~_|$]/i', $url_a))
{
  echo "<h3>";
  echo "Input value is not an url (need to begin with http:// or https://)";
  echo "</h3>";
}
else
{
  echo "<h3>";
  echo "Time:\n";
  print_r($date_time);
  echo "<br>";
  print_r("Main page: " . $url_a);
  echo "</h3>";
}
```

Listing 4.2: Check if the Web App URL is in HTTP format.

Next, the VT API is called to perform a scan of the main URL using CURL facilities (Listing 4.3).

```php
$scan_url_a = array('apikey' => $virustotal_api_key,'url'=> $url_a);
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, 'https://www.virustotal.com/vtapi/v2/url/scan');
curl_setopt($ch, CURLOPT_POST, True);
curl_setopt($ch, CURLOPT_VERBOSE, 1);
curl_setopt($ch, CURLOPT_RETURNTRANSFER ,True);
curl_setopt($ch, CURLOPT_POSTFIELDS, $scan_url_a);
$result=curl_exec ($ch);
$status_code_scan_vt = curl_getinfo($ch, CURLINFO_HTTP_CODE);
```

Listing 4.3: VT API scan request using CURL.

After performing the scan, a report is generated and stored in an associative array (Listing 4.4).

```php
$report_url_a = array('apikey' => $virustotal_api_key,'resource'=> $url_a);
$ch= curl_init();
curl_setopt($ch, CURLOPT_URL, 'https://www.virustotal.com/vtapi/v2/url/report');
curl_setopt($ch, CURLOPT_POST, True);
curl_setopt($ch, CURLOPT_VERBOSE, 1);
curl_setopt($ch, CURLOPT_ENCODING, 'gzip,deflate');
curl_setopt($ch, CURLOPT_USERAGENT, "gzip, My php curl client");
curl_setopt($ch, CURLOPT_RETURNTRANSFER ,True);
curl_setopt($ch, CURLOPT_POSTFIELDS, $report_url_a);
$report_vt_result_url_a=curl_exec($ch);
$report_vt_status_code_url_a = curl_getinfo($ch, CURLINFO_HTTP_CODE);
```

Listing 4.4: VT API report request using CURL.

In order to get relevant information from the response, we must decode and parse it from the returned array named `scans`. The important keys for the classification of the site are `result` and `detected`. If the value for the key `detected` is 1 then the `verdict` is true, which means that the site is malign (Listing 4.5).

```
$report_json_url_a = json_decode($report_vt_result_url_a, true);
$scan_url_a = $report_json_url_a['scans'];
$value = '';
$result = '';
$detected = '';
$veredict = false;

foreach ($scan_url_a as $value)
{
  $source = key($scan_url_a);
  $result = $value['result'];
  $detected = $value['detected'];

  switch($detected)
  {
    case 1:
      echo($source . " classification: <font color=red>" . $result . "</font>");
      echo "<br>";
      $verdict = true;
      break;
  }
next($scan_url_a);
}
unset($value);
```

Listing 4.5: Parsing the associative array to check the verdict.

The timestamp and verdict results will be stored in a database table for the Web App main URL analysis. After the analysis of the main Web App URL, another subroutine will crawl the main URL to find the external and internal links. The subroutine uses the DOM tree to find nodes with a tag `a` and attribute `href`. Using a regular expression it is possible to identify if the link found is external or not (Listing 4.6).

```
$dom = new DOMDocument()
@$dom->loadHTML($html);

foreach($dom->getElementsByTagName('a') as $link)
{
  $external_link = $link->getAttribute('href');

  if (preg_match('/\b(?:(?:https):\/\/|www\.)[-A-Z0-9+&@#\/%=~_|$?!:,.]*
               [A-Z0-9+&@#\/%=~_|$]/i', $external_link))
  {
...
```

Listing 4.6: Crawl the DOM tree to find the `href` attribute.

All the links classified as external will be sent to VT to be scanned and get a final report. The VT API will use CURL to get a JSON response. The returned report will be parsed to get the `key:value` result, as explained previously for the main Web App URL. Therefore, this result will be used to get the verdict benign or malign using the same process as for the main Web App URL. The external links found are stored on the database. For the internal links the tool will perform a CORS analysis. For this analysis it will *crawl* the DOM tree to find the tag `script` and attribute `src`. All the links on this node are parsed to find interesting strings related to CORS, namely the methods `cors`, `XMLHttpRequest`, `open`, `setRequestHeader`, `getAllResponseHeaders`, `send` and `readyState`. In Listing 4.7 we can see the piece of code responsible to detect this interesting strings.

```
$scripts = $dom->getElementsByTagName('script');

foreach($scripts as $script)
{
  $scriptSrc = $script->getAttribute('src');

  if ($scriptSrc!=' ')
  {
    $ch1 = curl_init($scriptSrc);
    curl_setopt($ch1, CURLOPT_URL, $scriptSrc);
    curl_setopt($ch1, CURLOPT_RETURNTRANSFER, 1);
    curl_setopt($ch1, CURLOPT_CONNECTTIMEOUT, 30);
    $html = curl_exec($ch1);

    $dom = new DOMDocument();
    @$dom->loadHTML($html);

    $usingCORS = substr_count($html, "cors");
    $usingXHRCount = substr_count($html, "XMLHttpRequest");
    $usingXHROpen = substr_count($html, "open");
    $usingXHRSetRequestHeader = substr_count($html, "setRequestHeader");
    $usingXHRGetAllResponseHeaders = substr_count($html, "getAllResponseHeaders");
    $usingXHRSend = substr_count($html, "send");
    $usingXHRReadyState = substr_count($html, "readyState");
...
```

Listing 4.7: Crawling for CORS strings.

A flag is used to identify CORS methods and print the result related to the detection. For the CA auditing engine, we employ three routines. One for the main, other for the external and one more for the internal links which are using a technique similar to the presented before.

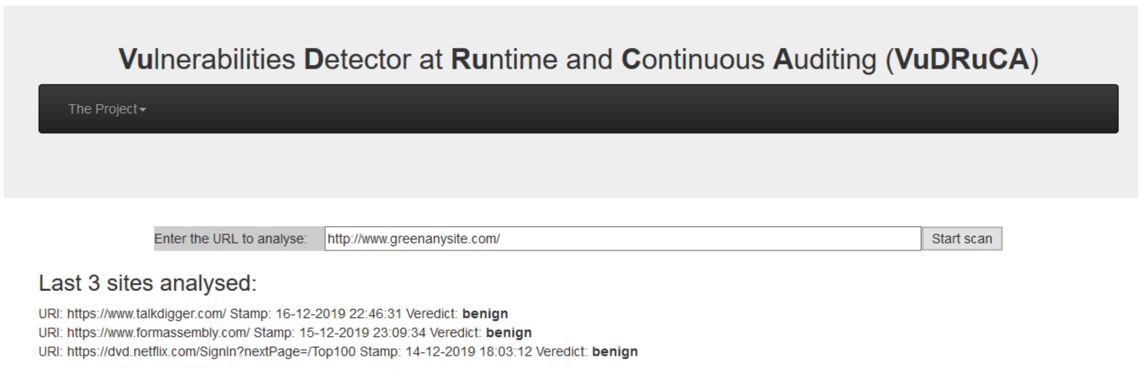
4.4 Example of Execution and Detection Case

This section presents an execution example, for the approach and architecture we propose. The user starts by introducing a URL of the Web App to analyse in the text box "Enter the URL to analyse" and clicks on the "Start scan" button (see Figure 4.5). The tool will automatically begin the crawling of the Web App to find external links and internal links. The main interface also presents the last three Web Apps that were analysed and their status, in this case benign (see Figure 4.5).

All the external links are sent to VT using a public API. When the analysis completes, the result is presented as in Figure 4.6. If the external site is not identified as malicious, the final verdict is benign (green), otherwise, it is malign (red).

The internal links are analysed to find interesting strings related to CORS. The results of this analysis are showed in Figure 4.7. The CORS functions found are identified in red color.

For the CA of the analysed Web App, the model uses a Task Scheduler routine every 24h as illustrated in Figure 4.8. This routine will check the Web App to find new external links and CORS vulnerabilities in the internal links. PHP scripts will run as programed in

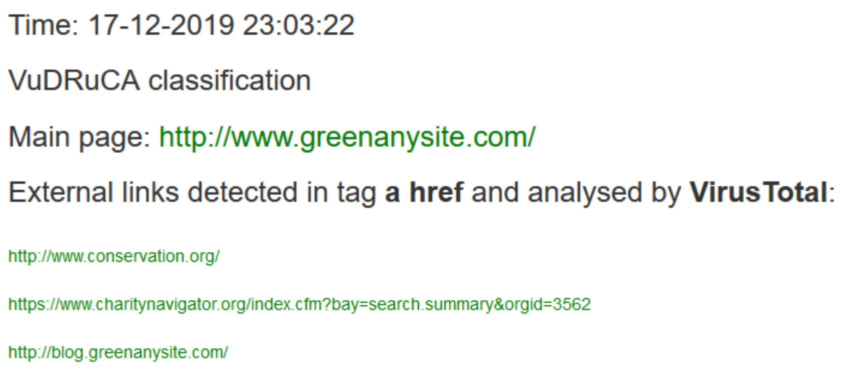

Vulnerabilities Detector at Runtime and Continuous Auditing (VuDRuCA)

Enter the URL to analyse: http://www.greenanysite.com/

Last 3 sites analysed:

URI: https://www.talkdigger.com/ Stamp: 16-12-2019 22:46:31 Veredict: **benign**
URI: https://www.formassembly.com/ Stamp: 15-12-2019 23:09:34 Veredict: **benign**
URI: https://dvd.netflix.com/SignIn?nextPage=/Top100 Stamp: 14-12-2019 18:03:12 Veredict: **benign**

Figure 4.5: Main interface.


Time: 17-12-2019 23:03:22

VuDRuCA classification

Main page: http://www.greenanysite.com/

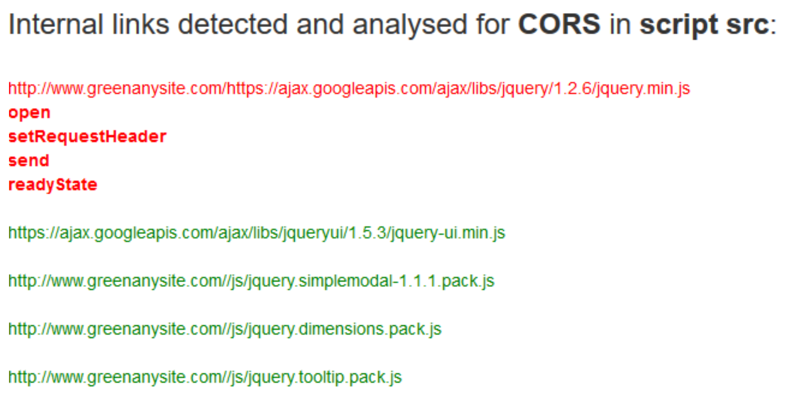
External links detected in tag **a href** and analysed by **VirusTotal**:

http://www.conservation.org/

https://www.charitynavigator.org/index.cfm?bay=search.summary&orgid=3562

http://blog.greenanysite.com/

Figure 4.6: Rep. and Ass. submodule results for the *https://www. greenanysite.com/*.


Internal links detected and analysed for **CORS** in **script src**:

http://www.greenanysite.com/https://ajax.googleapis.com/ajax/libs/jquery/1.2.6/jquery.min.js
open
setRequestHeader
send
readyState

https://ajax.googleapis.com/ajax/libs/jqueryui/1.5.3/jquery-ui.min.js

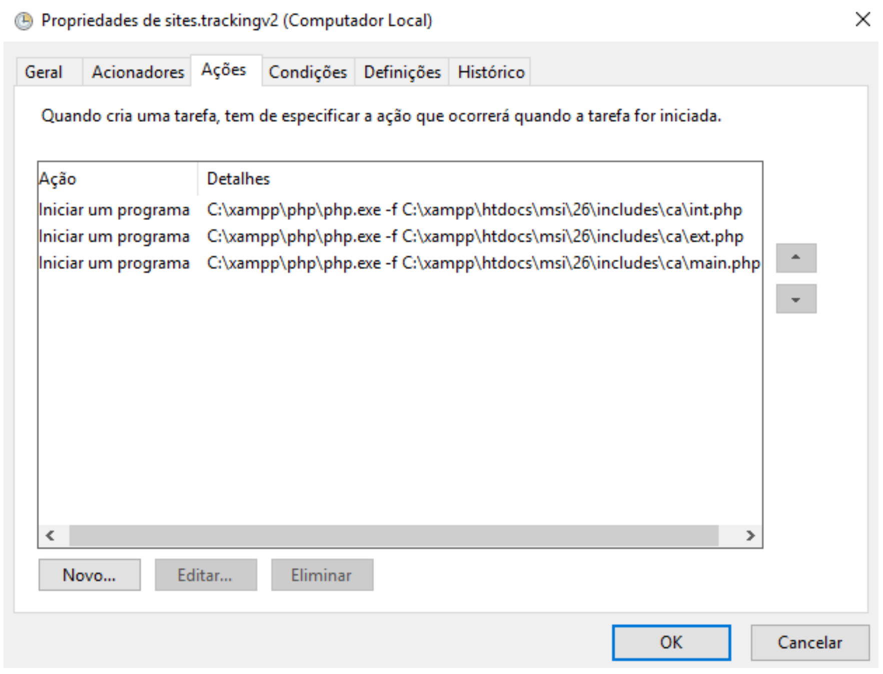
http://www.greenanysite.com//js/jquery.simplemodal-1.1.1.pack.js

http://www.greenanysite.com//js/jquery.dimensions.pack.js

http://www.greenanysite.com//js/jquery.tooltip.pack.js

Figure 4.7: Site CORS internal analysis result for the *https://www. greenanysite.com/*.

the task scheduler to perform a CA.


Propriedades de sites.trackingv2 (Computador Local)

Quando cria uma tarefa, tem de especificar a ação que ocorrerá quando a tarefa for iniciada.

Ação	Detalhes
Iniciar um programa	C:\xampp\php\php.exe -f C:\xampp\htdocs\msi\26\includes\ca\int.php
Iniciar um programa	C:\xampp\php\php.exe -f C:\xampp\htdocs\msi\26\includes\ca\ext.php
Iniciar um programa	C:\xampp\php\php.exe -f C:\xampp\htdocs\msi\26\includes\ca\main.php

Figure 4.8: Task Scheduler routine.

Chapter 5

Evaluation

The main objective of this chapter is to test and evaluate the capabilities of the VuDRuCA tool. The tool detects possible malicious external links and CORS invocation in the internal links of the Web Apps analysed. After the site's first analysis, it uses a CA approach to evaluate periodically both types of links and report their state.

For our purpose, a malicious link is related to a potential compromise of a Web App that is attacked using an injection of a malicious input, for example, an HTML injection that inserts a malicious instruction through the attribute `href` from the tag `a` or the attribute `src` on a tag `script` that points to a malicious site. Concerning this, it is supposed that the tool is capable of detecting if a Web App has external links pointing to malicious sites or is invoking Cross-Domain requests. Thus, some questions should be answered:

1. Are the external links of the Web App trustworthy?
2. Which internal links are using CORS?
3. Is the CA module efficient while analysing Web Apps?

5.1 AJAX Use-Case Characterization

There are some problems that developers can observe while building a Web App, such as a browser that refuses to access a remote resource. Usually, this happens when the Web App executes an AJAX Cross-Domain request with the jQuery interface, a Fetch API or a plain `XMLHttpRequest` (XHR). As a result, the AJAX request is not performed and data is not retrieved because the browser does not permit such access.

This issue is related to the security policy that defines the rules of how a web page can access an external resource (e.g., fonts, AJAX requests). Under the SOP, web browsers do not allow a web page to access resources whose origin differs from that of the current page. As explained before, the origin is considered to be different when the scheme, host-name or port of the resource does not match with the page. Overcoming the limitations of SOP security is possible using CORS.

CORS is a mechanism that defines a procedure in which the browser and the web server interact to determine whether to allow a web page to access a resource from a different origin. Nowadays, many of the AJAX sites use Cross-Domain requests. Also, the AJAX attack surface is larger than "normal" applications. All the typical attacks work against the AJAX interface, as it is made easier by the large amounts of client-side code that performs business logic and has to understand the application flow.

AJAX uses a XHR object for all communication with a server-side application, frequently a web service. A client sends a request to a specific URL on the same server as the original page and can receive any kind of reply from the server. These replies are often snippets of HTML, but can also be XML, JSON, image data, or anything else that JS can process. XHR objects retrieve the information of all servers on the web which could lead to various other attacks.

To evaluate VuDRuCA over our AJAX use-case, we created a list of 100 AJAX sites. The list was built based on three sources:

- *AJAX Goals* at: *"http://www.ajaxgoals.com/ajax-applications.html"*
- *BuiltWith* at: *"https://trends.builtwith.com/websitelist/AJAX-Libraries-API"*
- *Kiko* at: *"http://kiko.com/html/social/index.html"*

After collecting the 100 AJAX sites, we analysed all of them with our tool and the results are presented in the tables of Section 5.2. Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6, give an overview of these 100 Web Apps, whereas the remaining tables allow for a more detailed study of the behavior of the tool.

5.2 Experimental Phases

The evaluation was carried out in 3 phases. There was a first phase in which the 100 Web Apps were analysed regarding the existence of external and internal links. In the second phase, a subset of 30 Web Apps was selected to be categorized in relation to the number of external and internal links, and execution times. Finally, 10 Web Apps of the group of 30 were used to test the CA mechanism.

5.2.1 First Phase: 100 Web Apps

This phase was carried to verify the specificity of VuDRuCA. As described earlier, the tool was designed to identify external and internal links using crawling techniques. To demonstrate that it is possible the detection of malicious links, which are available for the user to click, only the `href` attribute was checked. From the tests it was observed that

Nr.	URL	External links VT analysis `a href`	Internal links CORS analysis `script src`
1	https://www.chegg.com/play/	Benign	Without Links
2	https://bobshideout.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; getAllResponseHeaders; send; readyState.
3	http://www.virtual-whiteboard.co.uk/	Benign	Without Links
4	https://www.formassembly.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; getAllResponseHeaders; send; readyState.
5	https://tiddlywiki.com/	Without Links	Without Links
6	http://www.greenanysite.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState.
7	https://www.talkdigger.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; getAllResponseHeaders; send; readyState.
8	https://study.com/	Benign	Without CORS
9	https://map.search.ch/	Benign	Without Links
10	http://www.sudokucraving.com/game.php	Without Links	Without Links

Table 5.1: List of 100 AJAX sites analysed by the VuDRuCA tool - 1/6 .

the implemented routine on VuDRuCA is able to find this type of attribute. In addition, for the detection of CORS, the tool looked into scripts to verify the presence of methods that make requests to remote resources.

From the 100 AJAX Web Apps reviewed, some are classified as not having external links in the sense that they do not resort to the `href` or `script` attribute. Some tests showed that Web Apps main URL can pass links using other attributes such as: `img`, `canvas`, `link`, `iframe`, `object`, `embed` and `link`. However, for the external links, these are not available for the user to click, thus being outside of our study. Regarding internal sites, the considered attribute was the `src`. This attribute was generally used for links and internal site functionalities.

The results are shown in Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6 (note that the table had to be divided in several subtables that are displayed in the following pages). If the tool found that the URL has external links, then the result from the VT analysis should indicate either *Benign* or *Malign*. If the Web App does not have external links then the result should be *Without links*. For the internal links, the tool lists the interesting strings associated with CORS, like `XMLHttpRequest`, `open`, `setRequestHeader`,

Nr.	URL	External links VT analysis `a href`	Internal links CORS analysis `script src`
11	https://www.t-mobile.com/	Withou Links	CORS; XMLHttpRequest; setRequestHeader; send; readyState.
12	http://www.rpad.org/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState.
13	https://roundcube.net/	Benign	Without Links
14	https://www.education.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState.
15	https://www.rapha.cc/eu/en/	Benign	Without CORS
16	https://www.protopage.com/	Benign	Without CORS
17	http://www.pressdisplay.com/pressdisplay/pt/Interstitial.aspx	Benign	Without CORS
18	http://www.pitstreet.com/cgi-bin/pitstreet/login.pl	Benign	Without Links
19	http://code.jalenack.com/periodic/	Benign	Without Links
20	https://dvd.netflix.com/SignIn?nextPage=/Top100	Benign	Without CORS

Table 5.2: List of 100 AJAX sites analysed by the VuDRuCA tool - 2/6 .

`getAllResponsesHeaders`, `send` and `readyState`.

If the tool detects CORS links, the tables present the XHR methods found. If the Web App has internal links but is not using CORS then the analysis is classified as *Without CORS*. Lastly, a Web App without internal links is displayed as *Without Links*. Overall, the results showed that there were no malign external sites discovered in the 100 Web Apps. For the internal links, there were 20 Web Apps that were using CORS methods, 17 that had internal links but without CORS methods, and the remaining 63 Web Apps had no links associated with scripts.

5.2.2 Second Phase: 30 Web Apps

After analysing the 100 Web Apps, the second phase processed a sample of 30 Web Apps. This selection is due to the fact that some sites do not use the elements required for the CA analysis and sites without links do not contribute to the intended analysis. We divided the 30 sites into different categories considering the execution times of the VuDRuCA analysis as well as the number of external and internal links found by the tool.

The results are presented in Table 5.7 and they show that the 30 Web Apps belong to 12 categories (Column 3) regarding technologies and services. VuDRuCA found 1258 external links, 238 internal links (Columns 4 and 5) and takes about 2563 sec. (43 min.) on average to analyse a site. Regarding the median, we observed that the tool takes 1097 sec. (18 min.) to analyze the site with 19 external links and 7 internal links. The standard deviation shows a significant variability on the time spent testing a site (3924 sec. or

Nr.	URL	External links VT analysis `a href`	Internal links CORS analysis `script src`
21	https://www.movietickets.com/	Benign	Without CORS
22	https://www.toysrus.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState.
23	http://www.xgames.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState.
24	http://mawisoft.com/	Benign	Without Links
25	http://www.informationsarchiv.biz/	Benign	Without Links
26	http://www1.xfiles.hotels-x.net/search/compare/hotels/index.html	Benign	Without Links
27	https://home.pandorabots.com/home.html	Benign	Without Links
28	https://www.google.com/maps	Benign	Without CORS
29	https://www.target.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState
30	http://paramoreredd.com/	Benign	Without CORS
31	http://htmledit.squarefree.com/	Benign	Without Links
32	https://www.dutchpipe.org/	Benign	Without CORS
33	https://1976design.com/blog/	Bening	Without CORS
34	http://digg.com/spy	Benign	Without Links
35	http://diegogiacomelli.com.br/	Bening	Without CORS
36	http://www.calendarhub.com/	Without Links	Without Links
37	http://www.hotels-balearic-islands.com/en/bandnews.html	Without Links	Without Links
38	https://del.icio.us/	Without Links	Without Links
39	https://www.foxsports.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState
40	https://www.amazon.com/	Benign	Without Links

Table 5.3: List of 100 AJAX sites analysed by the VuDRuCA tool - 3/6 .

65 min.). It is also possible to verify that each site analysed has a standard deviation of 63 external links and 6 internal links. For the 30 AJAX sites, we chose 12 categories regarding technologies and services used by the Web Apps.

The *Business* category contains sites that provide the latest business news on stock markets, financial and earnings. They give a view of the world markets in streaming, charts, stock tickers and quotes. *Career* is related to employment. The sites allow employers to post job requirements for a position to be filled by candidates. *Games* includes all gaming platforms such as betting, lotteries, and casinos. The *House* category includes household products and real estate. *Mobile* has mobile brandmarks and mobile operators. *Movies* is related to streaming movies platforms. *News* offer current news and opinions, such as those sponsored by newspapers and general-circulation magazines. *Photo* includes store and share photos platforms. The *Shopping* category contains web apps that

Nr.	URL	External links VT analysis `a href`	Internal links CORS analysis `script src`
41	https://www.enozom.com/	Benign	Without CORS
42	https://www.zillow.com/	Without Links	Without Links
43	https://finance.yahoo.com/tech/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState
44	https://www.flickr.com/	Without Links	Without CORS
45	https://search.yahoo.com/	Without Links	Without Links
46	http://www.aventureforth.com/?p=13	Without Links	Without Links
47	http://www.tagworld.com/	Without Links	Without Links
48	https://basecamp.com/retired/tadalist	Benign	Without CORS
49	http://sproutliner.com/	Without Links	Without Links
50	https://www.houzz.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState
51	https://www.bing.com/maps?FORM=LGCYVD	Without Links	Without Links
52	https://outlook.live.com/owa/	Benign	Without CORS
53	https://www.clickondetroit.com	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send;
54	https://24sevenoffice.com/uk/	Benign	Without Links
55	https://maps.a9.com/	Without Links	Without Links
56	https://www.objectgraph.com/	Without Links	Without Links
57	http://ajaxwrite.com/	Benign	Without Links
58	https://www.ebay.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send;
59	https://www.ask.com/	Without Links	Without Links
60	http://askalexia.com/	Benign	Without Links

Table 5.4: List of 100 AJAX sites analysed by the VuDRuCA tool - 4/6 .

feature on-line promotion or sale of general goods and services such as electronics, flowers, jewelry and music. It also includes on-line auction services such as eBay, Amazon, Priceline. *Social* represents platforms to build social networks or social relations among people who share similar interests, activities, backgrounds or real-life connections. *Sports* has web apps that pertain to recreational sports and active hobbies like fishing, hunting, jogging, as well as organized, professional and competitive sports. *Blogs* are platforms where a writer or a group of writers share their views on an individual subject. The results show that most of the Web Apps are categorized as *Shopping* and *Blogs* followed by *Social* category. From the analysis of Table 5.7 we can conclude that all categories use CORS which makes these Web Apps vulnerable, opening a range of attack surfaces.

Nr.	URL	External links VT analysis `a href`	Internal links CORS analysis `script src`
61	https://www.backbase.com/	Without Links	Without Links
62	http://www.bloxpress.org/	Benign	Without CORS
63	https://www.box.com/	Benign	Without Links
64	http://www.colr.org/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState
65	https://www.sephora.com/	Benign	Without CORS
66	http://www.toolani.de/	Without Links	Without Links
67	http://www.podcast-tuneup.com/	Without Links	Without Links
68	http://www.adworks.ro/	Without Links	Without Links
69	http://www.madebysofa.com/	Without Links	Without Links
70	http://www.monofactor.com/	Without Links	Without Links
71	http://www.alexbuga.com/	Without Links	Without Links
72	http://www.panic.com/coda/	Without Links	Without Links
73	http://www.arcinspirations.com/	Without Links	Without Links
74	http://www.dibusoft.com/	Without Links	Without Links
75	http://www.jasonjulien.com/	Without Links	Without Links
76	http://www.engageinteractive.co.uk/	Without Links	Without Links
77	http://www.jwhanif.net/	Without Links	Without Links
78	https://br.wordpress.com/	Benign	CORS; XMLHttpRequest; open; send; readyState
79	http://helldesign.net/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState
80	http://www.nebonmedia.com/	Without Links	Without Links

Table 5.5: List of 100 AJAX sites analysed by the VuDRuCA tool - 5/6 .

5.2.3 Third Phase: 10 Web Apps for CA

Finally, we selected 10 sites out of the 30 to be monitored in CA. We run VuDRuCA for 5 days with a CA every 24h and registered the results in Table 5.8. We considered the first day as the evaluation baseline, and in the subsequent days the checks were repeated to determine if the Web Apps suffer any changes.

To identify the Content Management Systems (CMS) used by the Web Apps that are analysed in CA, we employed the tool *Wappalyzer* that is a cross-platform utility that uncovers the technologies utilized on websites. For the sample we can conclude that WordPress is largely the most common CMS.

After reviewing the CA results over the 5 days, we found that there are some changes in the number of internal and external links in a few of the Web Apps reviewed. Although the differences found are not significant, we can see that internal and external links are introduced in Web Apps. For example, for the external links, we discovered that Web Apps 4, 39 and 50 suffered a change in the number of these links. The baseline of external links for Web App 4 was 16 on day 1 but on day 4 the number increased to 17. For site

Nr.	URL	External links VT analysis `a href`	Internal links CORS analysis `script src`
81	http://www.pikaboo.be/	Without Links	Without Links
82	http://www.mariusroosendaal.com/	Benign	Without Links
83	http://paramoreredd.com/	Benign	Whitout Links
84	http://www.mariusroosendaal.com/	Benign	Without Links
85	http://dragoninteractive.com/	Benign	Whitout Links
86	https://www.click2houston.com	Benign	CORS; XMLHttpRequest; open; setRequestHeader; send; readyState
87	http://www.cssmoon.com	Without Links	Without Links
88	http://www.playgroundblues.com/	Without Links	Without Links
89	http://kyanmedia.com/	Without Links	Without Links
90	http://www.komodomedia.com/	Benign	Without Links
91	https://www.click2houston.com	Without Links	Without Links
92	http://www.vitamin-j.de/	Without Links	Without Links
93	http://cliframework.com/	Without Links	Without Links
94	http://www.h4x3d.com/	Without Links	Without Links
95	http://www.designflavr.com/	Benign	Without Links
96	http://www.trashstars.com/	Benign	Without Links
97	http://www.authenticstyle.co.uk/	Withou Links	Withou Links
98	http://www.jedarecords.it/	Without Links	Without CORS
99	http://www.djfolio.com/	Withou Links	Without Links
100	https://br.wordpress.com/	Benign	CORS; XMLHttpRequest; open; setRequestHeader; getAllResponseHeaders; send; readyState.

Table 5.6: List of 100 AJAX sites analysed by the VuDRuCA tool - 6/6 .

39 the baseline was 67 external links, however, on day 2 we observed 71, on day 3 it was 70, and on day 4 it returned to 67. For Web App 50 the base is 230 and we observed 233 external links on day 2 and 236 on day 3. In these cases, the introduced links continue to be classified as benign. This result means that these links were intentionally introduced by Web Apps developers while performing updates. If results were obtained on external links classified as malicious, this could mean that malicious users injected code on the Web Apps.

For the internal links, only a change on Web App 78 was identified. The baseline for this Web App was 3 internal links but then it changed to 2 on days 2, 3 and 4. In this case, a reduction in the number of internal links should signify code updates. The CA evaluation also had the goal to show that Web Apps have their code constantly changed. However, being the Web App vulnerable to various types of injection attacks, this could lead to unintended changes. These are really the most dangerous and exploited by malicious actors.

To summarize, we can state that VuDRuCA is able to analyse external links passed in the `href` attribute of the tag `a` using the VT API. Of the analysed Web Apps, no malicious external links were found. Regarding the detection of internal links passed in

Nr.	URL	Category	Nr. External links	Nr. Internal links	Execution Time (sec.)
2	https://bobshideout.com/	Blogs	1	13*	62
4	https://www.formassembly.com/	Social	16	31*	967
6	http://www.greenanysite.com/	Shopping	3	6*	162
7	https://www.talkdigger.com/	Blogs	25	6*	1559
8	https://study.com/	Career	24	11	1363
11	https://www.t-mobile.com/	Mobile	-	13*	4
12	http://www.rpad.org/	Games	36	7*	2233
14	https://www.education.com/	Career	12	9*	709
15	https://www.rapha.cc/eu/en/	Shopping	42	12	2690
20	https://dvd.netflix.com/SignIn?nextPage=/Top100	Movies	3	7	183
21	https://www.movietickets.com/	Movies	8	5	497
22	https://www.toysrus.com/	Shopping	173	5*	10877
23	http://www.xgames.com/	Sports	60	4*	3187
29	https://www.target.com/	Shopping	5	7*	224
32	https://www.dutchpipe.org/	Web Development	1	1	31
33	https://1976design.com/blog/	Blogs	36	3	1725
35	http://diegogiacomelli.com.br/	House	21	5	1227
39	https://www.foxsports.com/	Sports	69	13*	4061
41	https://www.enozom.com/	Mobile	4	12	253
43	https://finance.yahoo.com/tech/	Business	32	12*	1757
44	https://www.flickr.com/	Photo	-	2	1
50	https://www.houzz.com/	House	230	4*	14442
58	https://www.ebay.com/	Shopping	227	9*	13989
62	http://www.bloxpress.org/	Web Development	5	2	282
64	http://www.colr.org/	Photo	4	10*	228
65	https://www.sephora.com/	Shoping	1	5	64
78	https://br.wordpress.com/	Social	73	3*	4620
79	http://helldesign.net/	Web Development	61	12*	3887
86	https://www.click2houston.com/	News	21	8*	1308
100	https://www.chasethetrend.com/	News	89	11	5657
Total			1258	238	76881
Average			42	8	2563
Median			19	7	1097
Standard Deviation			63	6	3924

Table 5.7: Characterization of 30 AJAX sites for CA (* Link with CORS).

the `src` attribute of the tag `script`, we can conclude that many of the Web Apps relax the SOP using the CORS technique. We can also affirm that the VuDRuCA CA module is efficient in the analysis of Web Apps, taking into account that it detects the alteration of the links passed in the attributes of the referred tags. Although a 24h period has been chosen for the period of the analysis, it can be fine-tuned according to the rigor required for the audit.

		Baseline				CA															
Day		**1**				**2**				**3**				**4**				**5**			
Nr.	CMS	Int. links	Exec. time	Ext. links	Exec. time	Int. links	Exec. time	Ext. links	Exec. time	Int. links	Exec. time	Ext. links	Exec. time	Int. links	Exec. time	Ext. links	Exec. time	Int. links	Exec. time	Ext. links	Exec. time
2	Facebook	13*	1	1	61	13*	5	1	61	13*	1	1	61	13*	2	1	61	13*	1	1	61
4	WordPress	34*	26	16	915	34*	24	16	916	34*	24	16	917	34*	21	**17**	976	34*	21	17	976
14	Facebook	9*	5	13	762	9*	4	13	763	9*	4	13	764	9*	4	13	762	9*	5	13	762
20	Bootstrap	7	1	3	185	7	1	3	183	7	1	3	184	7	1	3	186	7	1	3	185
35	Ruby on Rails	5*	2	21	1225	5*	3	21	1221	5*	3	21	1223	5*	2	21	1222	5*	2	21	1219
39	WordPress	13*	5	67	3851	13*	5	**71**	4091	13*	5	**70**	4030	13*	5	67	3851	13*	5	67	3851
41	Bootstrap	12	4	4	244	12	4	4	244	12	4	4	245	12	4	4	244	12	4	4	244
50	WordPress	4*	3	230	14044	4*	3	**233**	14226	4*	2	**236**	14414	4*	2	230	14044	4*	3	230	14042
78	WordPress	3*	2	73	4464	**2***	1	73	4469	**2***	1	73	4457	**2***	1	73	4464	3*	2	73	4460
100	WordPress	11*	2	89	5403	11*	4	89	5406	11*	4	89	5408	11*	3	89	5403	11*	2	89	5402
Total		111	51	517	31154	110	54	524	31580	111	51	517	31154	110	45	523	31213	111	51	517	31141
Average		11	5	52	3115	11	5	52	3158	11	5	52	3115	11	5	52	3121	11	5	52	3114
Median		10	3	19	1070	10	4	19	1069	10	3	19	1070	10	3	19	1099	10	3	19	1067
Standard Deviation		9	7	71	4309	9	7	72	4366	9	7	71	4309	9	6	71	4306	9	7	71	4309

Table 5.8: CA results execution for 5 days (* Link with CORS).

Chapter 6

Conclusion

This thesis allowed the development of the VuDRuCA tool that allows analyzing client-side code using HTML and JavaScript (JS) programming languages in Web Apps. Through crawling techniques, the tool analyzes possible compromises caused by injections attacks of the type HTML injection (HTMLi) in links to the external sites that are clickable by the user. It also identifies cross-origin calls used in AJAX technology to perform Cross-Origin Resource Sharing (CORS). This methodology is widely used in Web Apps today and can pose a threat if it is misconfigured or used, open an attack surface for malicious actors.

The use of the Virus Total (VT) commercial API presents some problems, namely when a large number of Web Apps are sent to analyse. In order to be able to analyze a large number of Web Apps, sleeps had to be used in the code of the application in order to cause a delay and give the VT time to be able to efficiently return the analysis of the Web App.

Related to the three phases for the VuDRuCA evaluation, we can conclude that in for the first phase, no malign sites were found in the external links passed in the tag `a` with the `href` attribute. Regarding the analysis of internal links, the routines used allowed the identification of CORS calls in Web Apps developed in AJAX. Of the 10 Web Apps, 20 were identified that resorted to CORS, 17 with internal links but that did not make CORS calls and 63 that did not have links in the `script` tag with the `src` attribute.

For the second phase, VuDRuCA identified 1258 external links and 238 internal links. The tool took 2563 sec. (43 min.) on average to analyze a Web App. The median obtained allows concluding that VuDRuCA takes about 1097 sec. (18 min.) to analyze a Web App with 19 external links and 7 internal links. The time shown is reasonable taking into account the aforementioned in relation to sleep to introduce a delay in the analysis and thus allow the sending of several Web App. The standard deviation obtained allows concluding that the data related to the execution times are spread over a range of values with a large amplitude in which the minimum execution time is 1 sec. and the highest is 14442 sec ..

For to the third phase, where 10 Web Apps of the 30 were selected to be monitored in CA, over a period of 5 days, with a periodicity of 24 hours, we found that there are some changes in the number of internal and external links, in some of the Web Apps. Although the differences found are not significant, we can see that internal and external links are introduced or removed.

For external links, we identified that Web Apps 4, 39 and 50 had changed. The baseline for Web Apps 4 links was 16 on day 1, but on day 4 the number increased to 17. Site 39, the baseline was 67 links, however, on day 2, we observed 71, on day 3, it was 70, and on day 4 it returned to 67. For Web App 50, the baseline was 230 and we observed 233 external links on day 2 and 236 on day 3.

In these cases, the links introduced continued to be classified as benign. This result means that these links were intentionally introduced by the developers when performing updates. If results were obtained with links classified as malicious, this could mean that malicious users had injected code into the Web Apps.

For internal links, only one change was identified in Web Apps 78. The baseline had 3 internal links, being changed to 2 on days 2, 3 and 4. In this case, a reduction in the number of internal links. The evaluation of the AC was essential to demonstrate that Web Apps and Sites have their code constantly being changed.

However, since Web Apps are vulnerable to various types of injection attacks and misconfigurations, unintended changes can happen and those are really the most dangerous.

6.1 Future Work

For future work, we believe that the results showed that the CA is a proactive approach that permits to automate the detection of changes in Web Apps and determines if they are associated with vulnerabilities. The CA allows the modeling of analytical data for subsequent monitoring to be taken into account instead of human judgment. Besides these advantages, it can generate alerts and produce reports frequently. In the future, the tool could be enhanced by integrating other malware analysis tools, such as Hybrid Analysis, Any.run or Analyz. Although VuDRuCA does not perform the analysis of `http` headers, this could be an additional feature to check if the server hosting the Web App is allowing CORS, for example, through additional fields such as `Access-Control Allow Origin`.

Bibliography

[1] "3 Ways to Exploit Misconfigured Cross-Origin Resource Sharing (CORS) pavan kumar," https://www.we45.com/blog/3-ways-to-exploit-misconfigured-cross-origin-resource-sharing-cors, accessed: 2019-10-15.

[2] "Cross-site scripting (XSS)," https://owasp.org/www-community/attacks/xss/, accessed: 2019-09-11.

[3] "Find security bugs," https://find-sec-bugs.github.io/, accessed: 2019-4-08.

[4] "Findbugs - find bugs in java programs," http://findbugs.sourceforge.net/, accessed: 2019-4-08.

[5] "Flawfidner home page," https://dwheeler.com/flawfinder/, accessed: 2019-09-18.

[6] "HTML5 top 10 threats stealth attacks and silent exploits," https://media.blackhat.com/ad-12/Shah/bh-ad-12-HTML5_Top_10_Shah_WP.pdf, accessed: 2019-12-12.

[7] "NIST 800-142 pratical combinational testing," https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf, accessed: 2019-07-10.

[8] "OWASP top 10 - 2017 the ten most critical web applications security risks," https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf, accessed: 2019-09-09.

[9] "Re-inforce programming security," https://en.wikipedia.org/wiki/RIPS, accessed: 2019-12-09.

[10] "Reflected cross site scripting (xss) attacks," https://www.imperva.com/learn/application-security/reflected-xss-attacks/, accessed: 2019-8-13.

[11] "Samy (computer worm)," https://en.wikipedia.org/wiki/Samy_(computer_worm), accessed: 2019-5-14.

[12] "Vulnerability assessment remediation," https://security.berkeley.edu/continuous-vulnerability-assessment-remediation-guideline, accessed: 2019-10-14.

[13] "Web application protection," http://awap.sourceforge.net/, accessed: 2019-12-09.

[14] "Your teammate for code quality and security," https://www.sonarqube.org/, accessed: 2019-4-08.

[15] "Federal office for information security, "BSI - study a penetration testing model"," 2018.

[16] "ISO/IEC 27005 - information technology - security techniques - information security risk managment," 2018.

[17] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Chainsaw: Chained automated workflow-based exploit generation," 2016, in Proceedings of the ACM SIGSAC Conference.

[18] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg, "Improving software security with static automated code analysis in an industry setting," *Softw., Pract. Exper.*, vol. 43, pp. 259–279, 2013.

[19] M. Bohme, V. Pham, and A. Roychudhury, "Coveragebased greybox fuzzing as markov chain," 2016, in Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.

[20] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," 2008, in Proceedings of the USENIX Conference on Operating Systems Design and Implementation.

[21] A. Camargo, "Painel técnico VI - auditoria contínua," 2012, in Seminário Controlos Internos and Compliance.

[22] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," 2012, in Proceedings of the IEEE Symposium on Security and Privacy.

[23] Chou and P.-H., "The pratical aspect of ajax security (chinese version)," 2008, master Thesis - Department of Information Managment - Shih Hsin University.

[24] R. F. Costa, "O futuro da auditoria no contexto dos enterprise resource planning," 2015, in Universidade de Aveiro.

[25] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution," 2013, in Proceedings of the USENIX Conference on Security.

[26] D. Evans and D. Larochelle, "Larochelle, d.: Improving security using extensible lightweight static analysis. ieee softw. 19, 42-51," *Software, IEEE*, vol. 19, pp. 42 – 51, 02 2002.

[27] ——, “Improving security using extensible lightweight static analysis,” *IEEE Software*, vol. 19, pp. 42–51, 2002.

[28] Farah, T., Shojol, M, Hassan, M, Alam, and D, “Assessment of vulnerabilities of web applications of bangladesh: A case studyof XSS and CSRF,” 2016, in Proceedings of the International Conference on Digital Information and Communication Technology and its Applications.

[29] M. Finifter and D. Wagner, “Exploring the relationship between web applications development tools and security,” 2011, in Proceedings of the USENIX Conference on Web application.

[30] E. Fong and V. Okun, “Web application scanners: Definitions and functions,” 2007.

[31] P. Godefroid, M. Levin, and D. Molnar, “Automated whitebox fuzz testing,” *Network and Distributed System Security Symposium*, pp. 151–166, 01 2008.

[32] G. Grieco, G. Grinblat, and L. Mounier, “Toward large-scale vulnerability discovering using machine learning,” 2016, in Proceedings of the ACM Conference on Data and Application Security and Privacy.

[33] T. Hofer, “Evaluating static source code analysis tools,” 2010.

[34] T. Jensen, H. Pederse, M. Olesen, and R. Hansen, “Thaps: automated vulnerability scanning of php applications,” 2012, in Proceedings of the Nordic Coference on Secure IT Systems.

[35] B. Johnson, Y. Song, E. Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” 2013, in Proceedings of the International Conference on Software Engineering.

[36] J. Jurn, T. Kim, and H. Kim, “An automated vulnerability detection and remediation method for software security,” *Sustainability*, vol. 10, p. 1652, 2018.

[37] J. King, “Symbolic execution and program testing,” 1976, commun. ACM.

[38] ——, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, pp. 385–394, 1976.

[39] G. Klees, S. Wei, and M. Hicks, “Evaluating fuzz testing,” 2018, in Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.

[40] J. Kronjee, “Discovering vulnerabilities using data-flow analysis and machine learning,” 2018, in Proceedings of the International Conference on Availability, Reliability and Security.

[41] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, “Sqlprob: A proxy-based architecture towards preventing sql injection attacks,” in Proceedings of the 2009 ACM Symposium on Applied Computing.

[42] I. Medeiros, M. Beatriz, N. Neve, and M. Correia, “Hacking the dbms to prevent injection attacks,” 2016, in Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy.

[43] I. Medeiros, N. Neves, and M. Correia, “Automatic detection and correction of web application vulnerabilities using data mining to predict false positives,” 2014, in Proceedings of the International Conference on World Wide Web.

[44] D. Muthukumaran, D. O’Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch, “Flowwatcher: Defending against data disclosure vulnerabilities in web applications,” 2015, in Proceedings of the ACM SIGSAC Conference on Computer and Communications Security.

[45] T. Oyetoyan, B. Milosheska, M. Grini, and D. Cruzes, “Myths and facts about static applications security testing tools: an action research at telenor digital,” 2018, in Proceedings of the International Conference on Agile Software Development.

[46] F. T. Pinto, “Auditoria contínua: Um novo paradigma de auditoria,” 2011.

[47] R. Russel, L. Kim, L. Hamilton, and M. McConley, “Automated vulnerability detection in source code using deep representation learning,” 2018, in Proceedings of the IEEE International Conference on Machine Learning and Applications.

[48] P. Saxena, S. Hanna, P. Poosankam, and D. Song, “FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications,” 2010, in Proceedings of the Network and Distributed System Security Symposium.

[49] J. Schwenk, M. Niemietz, and C. Mainka, “Same-origin policy: Evaluation in modern browsers,” 2017, in Proceedings of the USENIX Security Symposium.

[50] J. Smith, B. Johnson, and E. Murphy-Hill, “Questions developers ask while diagnosing potential security vulnerabilities with static analysis,” 2015, in Proceedings of the Joint Meeting on Foundations of Software Engineering.

[51] O. V. and B. P. Delaitre A., “NIST samate: static analysis tool exposition (sate) iv,” 2012.

[52] S. Washington, “Auditoria continua de dados como instrumento de automação do controlo empresarial,” 2012, master Thesis - São Paulo University.

[53] Y. Xie and A. Aiken, “Static detection of security vulnerabilities in scripting languages,” 2006, in Proceedings of the ACM SIGSAC Conference.