View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

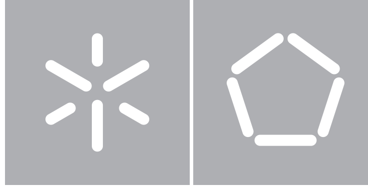
provided by Universidade do Minho: Reposito

Universidade do Minho
Escola de Engenharia

Luís Filipe Silva de Oliveira

Web Security Analysis
An approach to HTML5 risks and pitfalls

Outubro de 2013



Universidade do Minho
Escola de Engenharia

Luís Filipe Silva de Oliveira

Web Security Analysis
An approach to HTML5 risks and pitfalls

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor António Nestor Ribeiro

Outubro de 2013

I would like to thank my parents for all the support and for providing me the oportunity to be here.

I thank all my closest friends, specially Beatriz Almeida for all the support.

A special thanks for maincheck and to Joao Ribeiro, which gave me the theme, oportunity and help in every way possible.

Finally, I would like to thank my tutor, for the patient and support, and for helping me through this final stage of my academic life.

ABSTRACT

Web applications have become increasingly important in the last years. As more and more applications deal with sensitive and confidential data, the quantity and negative impact of security vulnerabilities has grown as well.

To understand this impact, this work presents a study over the most common and risky security vulnerabilities seen on web applications today. Also, with the emerging of the new HTML5 specification, an approach is made on how HTML5 will impact web security. The goal is to describe the new features of HTML5 and demonstrate how they may, or may not, introduce new security vulnerabilities to web applications.

Understanding the vulnerabilities is only the first step, as that knowledge is worthless if not applied into the software development life cycle. As so, this works performs a detailed analysis over static analysis tools, methodologies and strategies. Static analysis tools are very powerful, because they can help developers identifying possible vulnerabilities during all the development process.

Finally, this work compiles the information gathered in order to provide a set of guidelines on how static analysis tools need to evolve to face the new challenges presented by HTML5 and other emerging technologies. Also, a high level definition for the structure of a static analysis platform is presented.

As a whole, this work intends to be a complete survey over web security vulnerabilities, how they can evolve with the arriving of HTML5 and how can this be approached by static analysis tools.

RESUMO

Ao longo dos últimos anos, as aplicações web têm ganho especial importância. Com cada vez mais aplicações a lidar com informação sensível e confidencial, o número e o impacto das vulnerabilidades de segurança tem vindo a aumentar.

Para compreender este impacto, este trabalho fornece uma análise sobre as vulnerabilidades de segurança que introduzem mais risco e que são mais comuns nas aplicações web atuais. Com o emergir do HTML5, este trabalho pretende ainda estudar com principal cuidado, o impacto que as novas funcionalidades do HTML5 poderão ter na segurança das aplicações web. O objectivo é apresentar as novas funcionalidades do HTML5 e demonstrar de que forma estas podem, ou não, introduzir novas vulnerabilidades de segurança.

Compreender as vulnerabilidades é apenas o primeiro passo, porque esse conhecimento é inútil se não for aplicado durante o processo de desenvolvimento de software. Assim, este trabalho fornece uma análise sobre ferramentas, metodologias e estratégias para análise estática de código. As ferramentas de análise estática de código são extremamente poderosas, pois permitem identificar vulnerabilidades durante todo o processo de desenvolvimento de software.

Finalmente, este trabalho compila toda a informação reunida de forma a definir algumas diretrizes sobre como devem evoluir as ferramentas de análise estática de código de forma a conseguirem acompanhar com as alterações introduzidas pelo HTML5 e outras tecnologias emergentes. É ainda definido um modelo de alto nivel para a estrutura de uma plataforma de análise estática de código.

Concluindo, este trabalho pretende apresentar um estudo detalhado sobre vulnerabilidade de segurança em aplicações web, como estas podem evoluir com a chegada do HTML5 e como devem ser abordadas pelas ferramentas de análise estática de código.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

ABSTRACT iii

RESUMO iv

LIST OF FIGURES viii

CHAPTER

I. Introduction 1

1.1 Web 2.0 1
1.2 Motivation 2
1.3 Aims and objectives 3
1.4 Structure of the document 4

II. Web Security 5

2.1 Injection 7
2.2 Cross-Site Scripting 9
2.3 Broken Authentication and Session Management 11
2.4 Insecure Direct Object References 12
2.5 Cross-Site Request Forgery (CSRF) 14
2.6 Security Misconfiguration 15
2.7 Insecure Cryptographic Storage 16
2.8 Failure to Restrict URL Access 17
2.9 Insufficient Transport Layer Protection 18
2.10 Unvalidated Redirects and Forwards 19
2.11 Summary 19

III. How is web security evolving with HTML5 21

3.1 Browser Security Policies and Client-Side Vulnerabilities . . . 23
3.2 Introducing API and Vulnerabilities 25

3.2.1 CORS - Cross Origin Resource Sharing 27
3.2.2 Web Storage 29
3.2.3 Offline Web Application 32
3.2.4 Web Messaging 33
3.2.5 Web Sockets 34
3.2.6 New Tags 35
3.2.7 Web Workers 36
3.2.8 Sandbox 37
3.2.9 Drag and Drop 38
3.2.10 Geolocation 38
3.3 Summary 39

IV. Static Analysis: Background 41

4.1 Reviewing security 41
4.1.1 Dynamic Runtime Approaches 41
4.1.2 Static Analysis Approaches 42
4.1.3 Hybrid approaches 43
4.2 Building a Static Analysis Tool 44
4.2.1 Source code 45
4.2.2 Building models 46
4.2.3 Perform Analysis 52
4.2.4 Single Statement Assessment 52
4.2.5 Taint Analysis 53
4.2.6 Pointer Analysis 53
4.2.7 Generating reports 54
4.3 Open Source Tools 54
4.3.1 FindBugs 55
4.3.2 TAJ 55
4.3.3 Pixy 56
4.3.4 F4F 57
4.3.5 PMD 58
4.3.6 Yasca 58
4.3.7 PQL 59
4.4 Summary 60

V. An Approach to Handle Static Analysis 61

5.1 Basics 61
5.2 Features 63
5.2.1 Security Patterns 63
5.2.2 Context Sensitive 63
5.2.3 Modularity 64
5.2.4 Multi Language Analysis 65
5.2.5 Frameworks 67

5.2.6 Web services 69
5.2.7 Reporting 69
5.2.8 Software as a Service 70
5.3 Execution Process 70
5.3.1 Execution flow 71
5.3.2 Stage 1 72
5.3.3 Stage 2 73
5.3.4 Stage 3.1 74
5.3.5 Stage 3.2 75
5.3.6 Stage 4 77
5.4 Summary 78

VI. Conclusions 79

6.1 Output 80
6.2 Future Work 80

LIST OF FIGURES

2.1 Trend of new vulnerabilities found each year. 6

3.1 Browser/Plug-in vulnerabilities 2010-2012. 22

3.2 Browser architectural layers 24

3.3 Browser attack vectors. 25

3.4 HTML5 security overview by Compass, [*Michael Schmidt* (2011)] . . 26

4.1 High level model of a static analysis tool 45

4.2 Example of an abstract tree 48

4.3 Example of an abstract syntax tree 49

4.4 Example of a simple control flow graph. 51

4.5 Example of a simple call graph. 52

4.6 Example of a PQL query. 60

5.1 Kernel architecture 66

5.2 Global Workflow 71

5.3 Diagram of stage 1 72

5.4 Diagram of stage 2 73

5.5 Diagram of stage 3.1. 74

5.6 Diagram of stage 3 76

5.7 Diagram of stage 4 77

CHAPTER I

Introduction

1.1 Web 2.0

In the last few years, the internet has suffering a major revolution. A web application is no longer a set of pages where a user makes an HTTP request and the server returns a static HTML page as response. Web technologies and languages had evolved to enhance the interactivity between web applications and users, resulting in dynamic and very sophisticated web applications which work as service providers.

People use web applications to do all their day-to-day business operations online, and by doing that, they are trusting very sensitive information such as personal information, credit card numbers, on-line banking accounts, etc. People trust their web applications and this puts a great deal of responsibility on the service provider to guarantee a secure and safe interaction.

As people are moving their operations to the web, business companies moved their entire systems to the web. Most companies no longer have their own server and their own resources. Their systems and services are being moved to the web and we are watching to a paradigm-shift named cloud computing [*Lee* (2010); *Zissis and Lekkas* (2012); *Qian et al.* (2009); *IBM* (2009)]. Moreover, with the growing of cloud computing and the need for migrate more and more information to the cloud, web security becomes a major issue and is still making some companies reluctant in moving their systems to the cloud [*Dahbur et al.* (2011)].

1.2 Motivation

With the growing importance of web security, making secure software is critical, and although most of the applications deal with sensitive data, only a small fraction can afford a complete security review. Software review and testing are one of the most expensive stages in software development, and companies very often overlook this process.

Sometimes it's not because companies don't have the necessary time and resources to review the security of an application but because companies don't truly understand the risks and the importance which these reviews represent. Even with a properly security review, companies can't take security for granted.

Despite the great deal of attention given by the literature to web security, and the great improvement achieved, as web evolve and change so do hacking methods [*Cannings et al.* (2008)], new technologies are released and new systems are deployed every day, which makes the task of achieving total security almost impossible. The fact is that without changing anything in a secure application, that application can be compromised by a new threat never documented before.

All of this enforce the need for automated approaches to help developers in the process of building secure software. There are a great number of tools to perform this task, both open source and proprietary. This tools can, normally, be divided into two main categories: tools to use during development, before deploy (static analysis approaches) and tools to be used after deploy, with the system already running (dynamic runtime approaches). They work differently, use different strategies with different purposes and different results, but ultimately, all tools have the same goal: identify and help correcting security flaws.

The current changes on the web, are introducing new challenges for both static and dynamic approaches. With the growing complexity of web applications in the web 2.0, web applications are now built with multiple languages (both client and server-side), divided between multiple layers, and reuse public libraries and frameworks. The static analysis tools are having some troubles keeping up to this changes. Most of them simply parse and analyze one language, and do it sequentially, ignoring the complexity in the relations between layers, languages and frameworks.

A very strong example is the use of development frameworks. Using Java as example, it's inevitable the use of these frameworks (like Spring, Struts, JSF, etc) when building JEE web applications. They provide a simple and enhanced development environment for building web applications, making it easier to achieve high

performance, simplicity and scalability.

This enforce the need for a more evolved solution, capable of understand the relations and the complexity underlying a modern and complex web application. It's easy to find in the literature case studies which approach this needs, but in practice they never get implemented in real open source tools, and the prototypes released consist in very basic proof of concept implementations.

1.3 Aims and objectives

As described before, the number and importance of web applications is growing in an impressive rate. Following this growth is also the importance of those applications being secure, specially the ones dealing with sensitive data. As so, there is a common background to all the stages of this work: analyze the current state of security on modern web applications.

Drilling down on the subject, the first goal is to identify the most common categories of vulnerabilities present in web applications, understand how they have evolved over time and how they are changing with the current changes on the internet.

Among these current changes, there is a major subject of interest: HTML5. Day after day HTML5 is proving to be the near future of internet. As so, one of the most important goals of this work is to study the impact which HTML5 can have on the future of web security. So, the aim is to perform a complete analysis of the HTML5 specification, study each feature individually and understand which vulnerabilities they may, or may not, introduce into a web application.

Understanding the threats and vulnerabilities against web applications it's essential, but is worthless if that knowledge is not applied into the development process. Automated tools to analyze the integrity and security of an application are needed for both developers and security reviewers. As so, this work must present a detailed analysis on static analysis tools and methodologies.

It must be described the process of building a static analysis tool, covering all the stages of the process, and for each stage, identify the most common strategies, frameworks and methodologies used. Also, an analysis must be made over the existing open source tools, clearly identifying their strengths, weaknesses and strategies.

Then, a set of guidelines must be defined to approach how static analysis tools need to evolve in order to keep up with the constant changes and challenges faced by web applications. This guidelines include an analysis over what's missing on the current open source tools available. With those guidelines, a high level definition for

the structure of a platform capable of performing static analysis is presented. This platform must always look for scalability and for modularity but it must be mainly focused on Java and HTML5 languages.

To sum up, this dissertation intends to provide two main outputs: The first one is to provide a survey on the state of web security, to list and describe the current security vulnerabilities seen in web applications today, and to understand how they are evolving with the emerging of new technologies and methodologies, specially HTML5.

The second contribution is to define a set of guidelines on how static analysis tools need to evolve to keep up with the constant changes and challenges faced by web applications. Along with these guidelines, a proposal for a static analysis platform must be presented in the form of a high level model.

1.4 Structure of the document

In this introductory chapter, a contextualization of the problem was made and the main goals and guidelines for this work were defined and described.

Next, in chapter 2, a detailed survey over web security is made, identifying and defining the main categories of vulnerabilities seen on web applications today.

Chapter 3 shows how HTML5 affects the security of web applications, and what impact it can have on the vulnerabilities presented in chapter 2.

Then, in chapter 4, a background over static analysis tools, methodologies and strategies is presented, as well as examples of open sources static analysis tools available.

In chapter 5, the security knowledge from chapters 2 and 3, and the static analysis knowledge from chapter 4 is compiled together into a theoritical approach for a static analysis tool.

One final chapter shows the conclusions taken from this work, identifies the main difficulties and gives some pointers into future work.

CHAPTER II

Web Security

The growing integration of business systems into the web is moving more and more sensitive information into the web. Consequently, the number and the impact of security vulnerabilities on web applications is increasing in an impressive rate. A Symantec's security report [*Symantec* (2013)] published on April of 2013, shows that targeted attacks increased 42% in 2012, with a global average of 116 targeted attacks per day. A targeted attack is a specific kind of attack that is specially designed and has the specific goal of attacking one organization or industry. In the same report is shown that (see Figure 2.1) in 2012, it were identified 5291 new types of vulnerabilities, comparing to 4,989 of the year before and the 6,253 from 2010. Even with the general effort of all companies and entities in the software business, the general trend over time is to an increase of both the number of attacks and the number of vulnerabilities found.

Each vulnerability has different characteristics and can put at risk one application in many different ways. To measure that risk, it's important to understand what kind of attacks can exploit that vulnerability, and the impact which the attack can have on the system.

To rank that impact, a compilation of the ten most critical security vulnerabilities in web applications has been done in [*OWASP* (2010)]. This is a project from OWASP community, dated of 2010, and it's a reference when talking about the security of web applications. The ten vulnerabilities composing the top, ordered decreasingly by its risk, are the following:

- A1 - Injection
- A2 - Cross-Site Scripting (XSS)
- A3 - Broken Authentication and Session Management

Figure 2.1: Trend of new vulnerabilities found each year, *Symantec* (2013).

- A4 - Insecure Direct Object References
- A5 - Cross-Site Request Forgery (CSRF)
- A6 - Security Misconfiguration
- A7 - Insecure Cryptographic Storage
- A8 - Failure to Restrict URL Access
- A9 - Insufficient Transport Layer Protection
- A10 - Unvalidated Redirects and Forwards

To complete this approach and to better understand the attacks, during this chapter, it will be used a methodology from [*Cannings et al.* (2008)], in which the impact of an attack is characterized by three factors:

Popularity
: The frequency in which an attack is used: 1 being most rare, and 10 being widely used.

Simplicity
: The degree of skill that is needed to perform an attack which exploits the vulnerability: 10 being little or none, and 1 being great level of skill and experience.

Impact
: The potential damage which an attack can cause, if it is successfully performed: 1 being the revelation of non-important information about the target, and 10 being acquire superuser privileges or equivalent.

Now, this methodology will be applied into each one of the 10 vulnerabilities of the OWASP Top 10 presented above. Each vulnerability will be ranked within these three factors (Popularity, Simplicity and Impact). The grades and justifications presented below are based on the OWASP considerations for the Top 10 and in the analysis made by *Cannings et al.* (2008) when using this methodology.

2.1 Injection

Popularity: 9	Simplicity: 9	Impact: 10

Injection vulnerabilities [*Su and Wassermann* (2006), *Cannings et al.* (2008)] represent a major threat to the security of an application. This vulnerability can be exploited with common attacks like SQL injection, LDAP injection, XPATH injection or command injection.

Web applications are typically built in a two or more tiered architecture, consisting of, at least, one application running on a web server and a back-end database to store persistent data. When the input is not properly validated, and there is no strict separation between program instructions and user data, the attackers can sneak program instructions into places where the developer expects only to be data.

One common type of injection attack is *SQL injection*. SQL is the standard language for accessing databases and almost every application uses a SQL database to store persistent data. Because SQL syntax mix instructions with user data, the user data can be interpreted as instructions and be used to execute operations in the database. SQL injection attacks are very common, very simple to perform and can severely compromise a system. SQL Injection attacks can result in almost everything from data loss or corruption to a complete takeover of the database server. SQL Injection vulnerabilities are easier to detect when examining code, and more difficult via dynamic testing.

Another type of injection attack is *XPath Injection*. The problem is similar to SQL injection but, in this case, the sensitive data is stored in XML documents rather than a SQL Database. XPath it's a query language, created by developers for interpreting XML documents and, like SQL, XPath also has injection issues.

Like SQL injection, XPath injection attacks are relatively simple to perform, and can have a serious impact on the system. However, these kind of attacks are much less common than SQL injection attacks, because there are much more web applications using SQL Databases.

Another, well known and popular type of injection attacks are *command injection*. This kind of attacks happens when the attacker can inject system commands, and execute them in the server. This is more common in languages like PHP or PERL, and much less common in languages like Java or C#. A command injection attack takes advantages of improperly validated user input to inject system commands into the server. Then, in operations like system calls, those commands are executed, and can grant the attacker access, or even control of the server. Command injection attacks can have a serious impact on the system and are very simple to perform.

There are still a few injection attacks (like Buffer Overflows or LDAP injection) not described before. However, all injection attacks have a common property: all of them depend on the ability of the attacker to inject malicious data into critical places in the system.

One strategy to prevent against injection is to perform a good input validation (or sanitization). When validating input, it's important to constraint data types (e.g. if the input should be an integer, then it should only be accepted, if it is a valid integer), correctly escape strings (escaping special characters that can be used in a query or command), or even check the input against a list of known good values.

But, the very best strategy is to use specific API's. There are a few good API's which can, and should, be used to defend against injection. The most commons API's take advantage of a parameterized interface, to separate instructions from user data. A well-known and very effective approach against SQL Injection is *Prepared Statements*. Prepared statements are a very simple way to create SQL queries and are supported by all the main databases providers. When using prepared statements the database is able to distinguish between instructions and user data without jeopardizing performance. In fact, prepared statements were introduced as a performance optimization. However, this approach does not work so well when the queries are dynamically constructed (e.g. a page with different options that user can choose).

As a whole, injection attacks don't need a great level of expertise and can be performed from the user's browser in a simple and direct way. So, simplicity level is defined to 9.
Injection attacks were, for several years, the most popular and common web attack. However, in the last years, with the shift for the so called web 2.0 [*Cannings et al.* (2008)], and the rising of client-side technologies (e.g. AJAX, HTML5) XSS attacks become more and more common. Also, the biggest slice of injection vulnerabilities comes from SQL Injection which, with the use of prepared statements, decrease sig-

nificantly. However, and for multiple reasons, there are still a big number of web applications which don't use prepared statements. So, popularity level is defined to 9.
As defined in the [*OWASP* (2010)], "Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover" and so, impact level is defined to 10.

Because they are so dangerous, so popular and so easily exploitable, injection vulnerabilities are placed in the first place of the top 10.

2.2 Cross-Site Scripting

Popularity: 10	Simplicity: 8	Impact: 9

In the class of web security vulnerabilities, the most prevalent flaw is Cross-Site Scripting (usually abbreviated as XSS) [*Garcia-Alfaro and Navarro-Arribas* (2007); *Cannings et al.* (2008); *Louw and Venkatakrishnan* (2009); *Shanmugam and Ponnavaikko* (2008)]. XSS attacks occur when dynamically generated web pages display input which has not been properly validated. The name Cross-Site Scripting is derived from the fact that, an attacker can inject malicious scripts across security boundaries to a vulnerable, but trusted Website. XSS can be used to perform a very popular attack named phishing.

There are a variety of security controls and policies placed in web browsers to restrict JavaScript in a set of operations [*Jim et al.* (2007); *Cannings et al.* (2008)]. Also, browsers are always evolving and implementing new security controls. A successful XSS attack, depends on his ability to circumvent these controls and policies, and inject the malicious script into the browser. Sometimes, the security controls may work well but the security is compromised by exploring browser external plug-ins and features like Flash, Silverlight, Acrobat Reader, etc.

From all security controls and policies there are one main policy which underlies all the policies in all web browsers: the same origin policy. This policy defines that dynamic content, like JavaScript, only can interpret responses from the same origin where it came from. An origin is defined as a host name, a protocol and a port. The same origin policy it's essential because ensures that malicious websites cannot access other websites, and browsers only execute scripts from trusted websites. Later in this document, it will be explained how HTML5 directly breaks this security policy, and

what impact can this have in the future of XSS atttacks.

XSS are the most popular attacks on the web today, and don't require a great level of skills to be performed. Typically, if successful, a XSS attack does not result in compromising the user's machine. However, the attacker can use a XSS attack to steal cookies, hijack user sessions, temper with the website content, redirect users, introduce malware, etc.

XSS attacks are usually categorized into two categories: stored and reflected. Reflected (or non-persistent) are the most common XSS attacks. The malicious JavaScript is not stored on a server, and the user is lured into clicking in a link or submitting a form that immediately generates the malicious code (as part of an input) and send it to the vulnerable web server. The server reflects it back to the user's browser, which executes the code without any problem because it comes from a trusted web server.

In stored (or persistent) XSS attacks, the malicious JavaScript is stored in the vulnerable web server. A simple example is when the JavaScript is stored in a database: when the user request some information which is stored in the database, he will receive the malicious code instead, and his browser will execute it because it comes from a trusted web server.

With the rising of web 2.0, new XSS categories has emerged like DOM-based XSS. In this case, the attacker takes advantage of the Document Object Model (DOM) and the client-side JavaScript to execute the malicious code.

To prevent XSS attacks it's necessary to do a properly input validation, but this validation is not trivial at all. Like injection flaws, a possible strategy is to properly escape all input by constraint data types, correctly escape all input text or even compare the input to a list of known good values.

But Cross-Site Scripting is a little more complex to prevent than injections. As stated before, some XSS attacks can take advantage, for example, from external browser plug-ins and features.

Much like injection, XSS attacks don't need a great level of expertise, and can be performed directly from the user's browser. So, simplicity level is also defined to 8.
Popularity level of XSS attacks is defined to 10. XSS attacks not only are the most popular and common attacks on the web today, as they will become more and more popular with the emerging of technologies like HTML5.
As defined in the [*OWASP* (2010)], "Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users,

hijack the user's browser using malware, etc.". However, the impact is not as severe as in injection and so, the level of impact in XSS vulnerabilities is defined to 9 and they are placed in the OWASP Top 10 as the second most risky vulnerability.

2.3 Broken Authentication and Session Management

Popularity: 8	Simplicity: 6	Impact: 10

The process of authentication plays an essential role in a web application, defining which information and services the user is able to access and what is the role of the user in the system. Authentication is normally performed by asking the user for a username and a password.

Session management [*OWASP* (2013c)] is another very important process and often overlooked in the development. Sessions are maintained by the server, which creates a session for each client interacting with it. When a server receives a new HTTP request from a client, a session ID is generated by the server and sent back to the client within the HTTP response. This session ID uniquely identifies the client interacting with the server. After this first connection, the session ID is always passed within all requests sent and received between client and server. Session management includes all tasks made in both client and server sides to establish and maintain the sessions active and secure.

To prevent authentication vulnerabilities [*OWASP* (2013a)] is very important to implement strong password and password recovery mechanisms. A strong password worths nothing, if an attacker is able to gain access to an account with the password recovery mechanism. The error messages can also tell an attacker much about an account. As so, error messages must be generic and not give the attacker any leads about what went wrong. If fact, this is not valid only for authentication proposes. Error messages should be treated with care in all applications contents as they can give an attacker useful leads to system vulnerabilities. The user should also be granted with a limited number of attempts to correctly insert his credentials. If not, a brute force attack could be performed. Another important measure is to store all passwords with a strong and secure encryption. If not, and if an attacker gains access to the system's password database (e.g. with an SQL injection attack), the attacker would have access to all user's accounts and data.

All the communications between server and client are made via HTTP, a stateless protocol. As so, all the session management mechanisms have to be implemented by the developer, and it becomes its responsibility to ensure the security of the process and to prevent session management vulnerabilities [*OWASP* (2013c)]. To achieve this, a secure session management mechanism must start with an efficient generation of session identifiers. The name of the session ID should not be descriptive as it should not provide any information about the session ID itself, the system or implementation details. Much like a strong password, a session ID must be long enough to prevent brute force attacks and must have a random component to make it impossible to guess with statistical analysis techniques. Also, the content of the session ID must be restricted to the information which is absolutely necessary to identify the client, and should never include any kind of sensitive information about the user, the system or even the operation in progress.

It's very common for developers to build their own authentication and session management systems and, consequently, many of this systems have vulnerabilities. However, as each implementation has its own characteristics, finding flaws is not trivial at all. In fact, finding this kind of vulnerabilities can be very difficult and only possible to attackers with a certain level of expertise. So, the simplicity level is awarded to 6.

The impact which this vulnerabilities can have on a system is very severe. If an attacker is able to steal user credentials, or access the system on behalf of another user, the attacker can do anything the victim could do and access all of the victim's information. The consequences become even more severe if the attacker successfully perform this attack in sensitive system like banking or e-commerce, or if the attacker get access to privileged accounts. As so, the impact level is defined to 10 and they appear on the third place of the Top 10.

Authentication and session management attacks are not as common as XSS or SQL injection, mainly because this vulnerabilities are not so simple to exploit. However, getting access to user's credentials and information is one of most desirable aims for attackers, and so popularity level is defined to 8.

2.4 Insecure Direct Object References

Popularity: 7	Simplicity: 9	Impact: 8

In a communication between server and client, there are multiple parameters

which allows the client to send application data within the request to the server, and the server to understand the request. If a web application exposes an internal implementation object to the user, by directly reference it in one of this parameters, an attacker can modify it in an attempt to access objects which he doesn't have permission for. This references can be present either as URL or form parameters, and can reference files, directories, database records and primary keys.

A very common vulnerability in the category of Insecure Direct Object References, is Directory Traversal [*Cannings et al.* (2008)]. If a web application open files based on parameters, the attacker can deliberately change this parameters to read arbitrary files on web servers.

Another very common example within this category of vulnerabilities is Open Redirects [*Cannings et al.* (2008)]. If the web application has a parameter which allows the user to be redirected to another page, the attacker can change this parameter and lure potential victims to a malicious website of his choice.

To prevent Insecure Direct Object References, developers should use indirect references and limit the mapping to values authorized for each user. A white list defining which objects the user is allowed to access is a simple and effective solution. By reference the objects indirectly, the developer is not exposing names, URL's or database keys, making the attacker task much more difficult.

Direct mapping can easily be guessed by attackers and be performed by any user within his browser with simple parameter tampering. The attacker just have to understand a little about the logic of web applications and he will easily guess paths and variables that will lead him to sensitive information. If that information is not properly saved with authentication methods, the attack will be succeeded. So, simplicity level is defined to 9.

The risk introduced by this vulnerabilities depends on the information that the attacker is able to find. A successfully attack will may be able to steal sensitive information which he was not suppose to have access to, but hardly will become in a DOS attack or have consequences to the application performance. Consequently, it receives 8 for impact.

Attacks over insecure direct object references are not as popular as the ones presented above, but due to the fact that they don't need a great level of expertise and they can lead to sensitive information, they are still very common and receive 7 for popularity.

2.5 Cross-Site Request Forgery (CSRF)

Popularity: 8	Simplicity: 7	Impact: 9

Cross-site request forgery (usually abbreviated as XSRF or CSRF) allows a malicious website, visited by the victim, to perform actions in a vulnerable application, taking advantage of the victim's access and privilege level on that application.

CSRF attacks leverage from the inherent statelessness of the web applications and authenticated users (recall that for authentication proposes, the user's browser store a session ID and that session ID is sent in all the requests to and from the server). So, the malicious website forces the victim's browser to make a GET or POST HTTP request to the vulnerable web application, with query parameters or form field values and, consequently, the victim's browser will send the authentication credentials along with the request, and the attacker will be able to perform actions in behalf of the victim. Of course that, for this attacks to be succeeded, the user has to be authenticated.

It's intuitive to think that the same-origin policy (explained later in chapter 3) will not allow this to happen. In fact, the same-origin policy makes it impossible to the attacker's malicious website to read any data related to the victim's account, but, the attacker is still able to blindly cause the victim's browser to make the HTTP request, which is enough to perform the attack.

Preventing CSRF is done by including a unique token in the body of each HTTP Request. The token should be unique for each HTTP request or, at least, for each user session. The browser has the session ID stored, and sends it in the request, but, does not have the secret token, and so, the request will go without it and the server will be able to reject the request. The secret value can be a session ID stored in a client-side cookie or a random value generated from a cryptographically secure random number generator.

There are more rudimentary ways to prevent CSRF like, for example, every time the user performs a sensitive operation, the system can ask the user for his password. Off course this is not the most pleasant way, but it is a very strong policy and can be a good strategy to enhance security for very-sensitive operations like changing password, banking transactions, etc.

Despite the extremely dangerous nature of CSRF attacks, during several years, these attacks have received less attention than the more easily understood web application vulnerabilities such as XSS or Injection. However, in the last few years, they

began to be taken seriously and listing in the OWASP Top 10 project was achieved in 2007. Even in 2007, they entered directly into the fifth place and they maintained their position in the 2010 top. With a successfully CSRF attack, the attacker is able to perform any action that the victim can perform, and change any information that the victim is allowed to change. As so, impact level is defined to 9.

CSRF vulnerabilities are relatively simple to exploit. However, the attacker still needs a little knowledge to build a malicious website, and the ability to lure the victim into access it. Simplicity level for CSRF is then defined to 7.

Despite the fact that CSRF is very easy to prevent and detect, there are still a great number of web applications vulnerable to it. Consequently, CSRF attacks are very common and there are still reports about this kind of attacks targeting big and relevant systems. As so, popularity level is defined to 8.

2.6 Security Misconfiguration

Popularity: 7	Simplicity: 6	Impact: 10

Security Misconfiguration vulnerabilities are the result of configuration weaknesses found in web applications. This category of vulnerabilities is very easy to understand, but includes a very wide range of possible flaws. A simple misconfiguration can seriously compromise the security of a web application and it can be present in any level of an application. A misconfiguration in an application level framework can introduce a vulnerability in the same way as a misconfiguration in a web server can.

To prevent this kind of attacks it's important to keep all systems up to date (including OS, Web and application servers, DBMS, frameworks, libraries, etc). Also, regular verification and testing is needed.

It's important to carefully configure remote server accesses. Remote interfaces present an attractive window to opportunistic attackers. A misconfiguration in the process and the window become even bigger.

A very dangerous misconfiguration is the use of default or weak passwords and incorrect permissions set. Default passwords are public and should never be used. It's essential to use strong passwords and a well defined permission scheme.

Another very important configuration is error handling. Error messages in a web application should be treated with care and should not give any useful information to the attacker about the system. The less the attackers knows, the lower is the probability of him successfully finding a vulnerability in the system.

Many applications come with unnecessary and unsafe features. Unnecessary features must be disabled and avoided. If an unsafe feature is really necessary, contra-measures must be implemented.

Misconfiguration may provide means for an attacker to bypass authentication methods and gain access to sensitive information and places, perhaps with elevated privileges. Misconfiguration can even result in a complete system compromise. So, impact level is defined to 10.

To take advantage of poor configurations, a great level of skill is needed. To successfully perform an attack exploiting this vulnerabilities, the attacker needs to understand all the technologies (frameworks, servers, libraries, etc) in which the application is built. Then, he needs to understand the strengths and weaknesses of the technology he is exploiting, and understand where are the possible flaws that may give him a way in. On top of this, the attacker needs skill and knowledge to perform the attack. Simplicity level is, then, defined to 6.

Mainly because these vulnerabilities are harder to exploit then the previous ones presented, they are not so popular. They are very dangerous, but they need a greater level of expertise and time, and different applications need different approaches. As so, popularity level is defined to 7.

2.7 Insecure Cryptographic Storage

Popularity: 6	Simplicity: 6	Impact: 8

Strong cryptography is a critical piece of information security that can be applied at many levels, from data storage to network communication. One of the most common categories of security problems is the misapplication of cryptography in storage. If an attacker gains access to some sensitive information and that information is stored without encryption, the attacker will have no problem reading it. Developers and administrators sometimes make the deceptively assumption that their system is 100% secure, and so, encryption is not so important. There are no totally secure systems, and in the event of a successful attack, the attacker will be able to easily read non-encrypted information. On the other hand, with a proper encryption, even with a successful attack, the sensitive data may not be compromised as the attacker may not be able to decrypt it.

Cryptography it's an area which can look deceptively easy, when in reality there are an overwhelming number of pitfalls. It's a very wide area and, in the context of

this work, will not be specified with much detail.

To prevent this vulnerabilities to happen, it's essential the use of secure and standard encryption algorithms. Also, the encryption keys and passwords used must be secure and can not, in any way, be compromised.

Insecure cryptographic storage may lead to the compromise of sensitive information. But, that's not possible if the attacker don't get to the information first, exploiting another flaw. So, impact level is defined to 8.
Taking advantage of this kind of vulnerabilities directly depends on how insecure is the encryption. If there is no encryption, then no skill is needed. As the encryption quality increases, so the skill needed to perform the attack. Simplicity level is, then, defined to 6.
The attacker normally tries to get the information first, not knowing if it is properly encrypted or not. So, the attacker don't exactly exploit this vulnerabilities. On top of that, this vulnerabilities don't are very common and so, popularity level is defined to 6.

2.8 Failure to Restrict URL Access

Popularity: 5	Simplicity: 10	Impact: 8

This category of vulnerabilities occur when developers don't properly check if the user has permission to access a particular page. If not, an attacker can easily change the URL to a page which he does not have permission for, and access the information like an authorized user.

To prevent this vulnerability, a developer just need to guarantee that, for each page, a verification of the user access level is done. If the user is not allowed to open the page, then, the request must be blocked. For this to be done efficiently, it's important to have a well defined permissions scheme which identifies, clearly, the access level for each user.

Exploiting flaws of non-restricted URL access is very simple. The attacker just needs to change the URL to the page he doesn't have permission for and, if the vulnerability exists, he will be granted with access immediately. So, simplicity level is defined to 10.

This is a very well understood set of vulnerabilities. Developers understand exactly what can happen if they don't check before granting access to a page. So, the

number of this vulnerabilities is really low and consequently, the number of attacks performed is also very low. Popularity level is defined to 5.

In a successfully performed attack, the attacker gains complete access to the information and functionalities that a user with more permissions would have. So, this kind of vulnerabilities can seriously compromise sensitive and confidential information. Impact level is defined to 8.

2.9 Insufficient Transport Layer Protection

Popularity: 8	Simplicity: 5	Impact: 9

Insufficient transport layer protection can introduce vulnerabilities in two scenarios: when malicious traffic are interpreted by the web server, or when network packets, not properly secured, are intercepted by an attacker.

Malicious traffic consists in data packets that, when interpreted by the server and executed in the software, can compromise sensitive information, the server execution or even the user's machine.

To prevent this scenario, proper tools and protocols must be used. Firewalls and intrusion detection systems (IDSs) are two types of tools that must be used to deal with potentially malicious network traffic, both on client and server sides.

SSL(Secure Sockets Layer) protocol must be used to secure communications between client and server. SSL allows the client and server to communicate over an encrypted channel with message integrity in which the client can authenticate the server. Frequently, developers use SSL only during authentication, what can compromise the security of sensitive data, including session IDs. So, a web application must, at all times, require a well-defined SSL for all sensitive pages.

The server certificate is also very important because they help protecting users from deceitful activities. Basically, the certificate is a proof you show to the user to ensure him that you are who you're saying, and not someone else posing as you. So, the server certificate must be legitimate, well configured and up-to-date.

To explore this vulnerabilities, an attacker needs to understand, not only from web applications architecture, but also from network protocols and technologies. Of the OWASP Top 10, this is probably the set of vulnerabilities more difficult to exploit. As so, it receives 5 for simplicity.

Transport layer vulnerabilities can expose user's sensitive and confidential information. If the attacker accesses to an administrator's account information, the system

can be compromised. As so, impact level is defined to 9.

Exploits for this category of vulnerabilities are very common. Besides being hard to exploit, they are very dangerous and so, become very tempting for possible attackers. As so, popularity level is defined to 8.

2.10 Unvalidated Redirects and Forwards

Popularity: 6	Simplicity: 8	Impact: 7

Web applications regularly redirect users between pages (internal or external). This can introduce a vulnerability, if the destiny page is specified in a parameter defined by the user. Consequently, an attacker can very easily temper with the parameter and redirect potential victims to a malicious page.

To prevent this vulnerabilities, the best approach is definitively not using redirections coming from parameters. There are very few situations where this solution is needed. However, if the situation strictly requires this solution, then a proper validation must be applied verifying if the the destiny is valid and authorized for the user. A white list verification, with a comparison against a list of authorized destinies is a very good solution.

The required skill to exploit this vulnerabilities is not very high. An attacker just need to modify the parameter to the URL of his choice in order to redirect the victim into the malicious page. The victim trusts the application and, so, he will not get any suspicious about being redirected. The simplicity level is defined to 8.

These category of vulnerabilities is not very common. Mainly because there a few case scenarios where a parameter based redirection is needed. As so, popularity level is defined to 6.

The impact which an attack exploiting this vulnerabilities can have, depends on the malicious website for which the victim will be redirected. The malicious page can do anything from installing malware until convincing the victim into grant confidential information. The attack just works as a path to make the victim visiting the malicious page. Nothing really happens in the application. So, impact level is defined to 7.

2.11 Summary

In this chapter, it was completed the first goal defined to this dissertation: identify the most common categories of vulnerabilities present in web applications.

This was made using the ten vulnerabilities presented by OWASP in their Top 10 of 2010 and a methodology from *Cannings et al.* (2008). Following this methodology, each category of vulnerabilities was evaluated based on three factors: popularity, simplicity and impact. Were also described some attacks which each vulnerability can lead to and some pointers on how to avoid them.

The next step is to study the HTML5 specification, and understand how the new features can impact the vulnerabilities described in this chapter.

CHAPTER III

How is web security evolving with HTML5

Most of the web applications seen today are built in HTML4, the standard introduced in 1997. But, in the last decade, the web and the way we use it has changed dramatically. With the growing need for interactivity in web 2.0, web applications logic is significantly shifting from the server to the browser.

"With the inability of the HTML4 to follow this advances, web developers tried to fill the gap by adopting a variety of new technologies that use a combination of Javascript, DOM and XHTML communication, or by including third-party plug-ins like Flash or Silverlight.

As a successor of HTML 4.01 and XHTML 1.1, is now emerging the Hypertex Markup Language, version 5 (HTML5). It was founded by the Web Hypertext Application Technology Working Group (WHATWG) in 2004, but only later named of HTML5. But, HTML5 is not a common upgrade, it's actually a whole lot of new individual features to enrich and enhance web applications", [*Michael Schmidt* (2011)].

With this change from server to browser processing, browser security is gaining even more relevance. Browsers are relatively secure, and in general, the number of security vulnerabilities found in web browsers is dropping every year. Still, a big number of vulnerabilities were reported during the last year in the most popular web browsers. Furthermore, there is another critical entrypoint for security vulnerabilities, the so called third-party plug-ins.

In Figure 3.1, it's presented a comparison based on the percentage of vulnerabilities found during the last three years, for the most popular browsers and plug-ins.

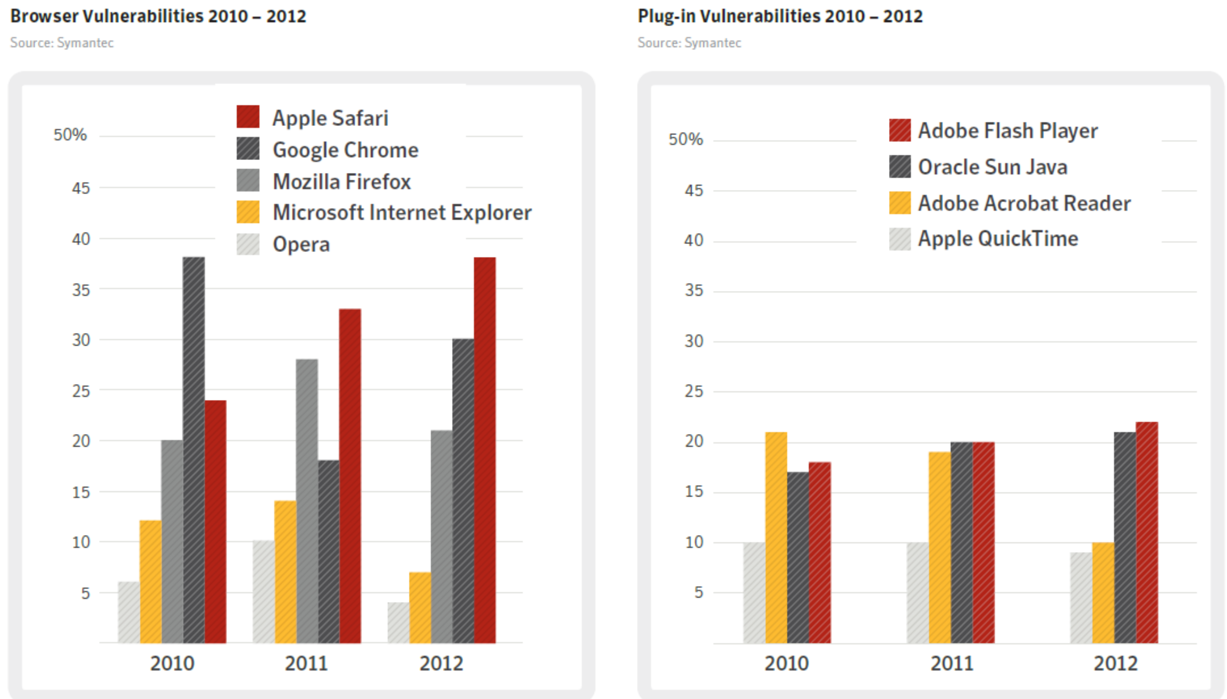


Figure 3.1: Browser/Plug-in vulnerabilities 2010-2012, [*Symantec* (2013)].

The role played by browsers in web security will grow even more over time, as HTML5 is being implemented. Moreover, and despite its great acceptance, concerns about the security of the new features of HTML5 persist. Like in every new technology, new features bring new security concerns and can be the root cause for new attack vectors. XSS attacks will get even more relevance and the browser will definitely be the most critical point when addressing web security.

In this chapter, the main goal is to analyze the new HTML5 specification and the implications it can have on web security. For each of the most important HTML5 features, it will be given a little description about its usefulness followed by an analysis over the security consequences it can bring. In addition to bring new security vulnerabilities, HTML5 will impact some of the existing ones.

In chapter 2, an analysis over the most popular categories of vulnerabilities was made. But, the analysis was mainly focused on a set of categories usually known as server-side vulnerabilities. So, this chapter will start by filling the gap on another perspective: client-side vulnerabilities, which can compromise the user's browser and computer, but usually have no consequences over the application's behavior or information.

Also in this chapter, in order to better understand the concept and the consequences of client-side vulnerabilities, it will be made an analysis over a typical browser architecture and the main security policies surrounding it. The fact is that, when evaluating the new HTML5 specification and his security risks, a web browser takes special importance. In fact, most of the new features security is browser responsibility. Browser vendors are working in the implementation and they are doing it with extreme care, but like always, they are doing it independently from each other and the same features work differently in different browsers.

That's not a goal of this chapter to prove, in any way, that HTML5 is insecure. In fact, HTML5 was designed with integrated security, probably like any language before. In the language specification [1], each feature has its own subsection with security considerations, describing how the feature must be implemented and what are the consequences of a poor implementation. But, at the same time, the language is going to be complex and support a wide variety of functions that are currently handled by multiple plug-ins. This sort of broad functionality, in general, tends to raise the potential for attacks.

3.1 Browser Security Policies and Client-Side Vulnerabilities

The shift of the application business from the server to the browser implies a shifting in the technologies used.

Technologies like Flash or Silverlight, that are around longer, have been integrated into the browser as external plug-ins. On the other hand, HTML5 features are directly integrated into the browser, and don't need any plug-in to run. Eventually, it should be expectable from all browsers to have a similar set of features but, for now, browsers are in very different stages of development. As so, browser compatibility is, and will continue to be a significant issue regarding HTML5. For now, the HTML5 specification is in the development stages and, some browsers may choose not to adopt a specific API right away (due to instability, ambiguity or other issues with the specification).

Every change made in the technologies used can impact the browser's model, and can introduce possible security faults. Figure 3.2 represents a browser model:

[1]HTML5 Language Specification, 2013: "http://www.w3.org/TR/html-markup/"

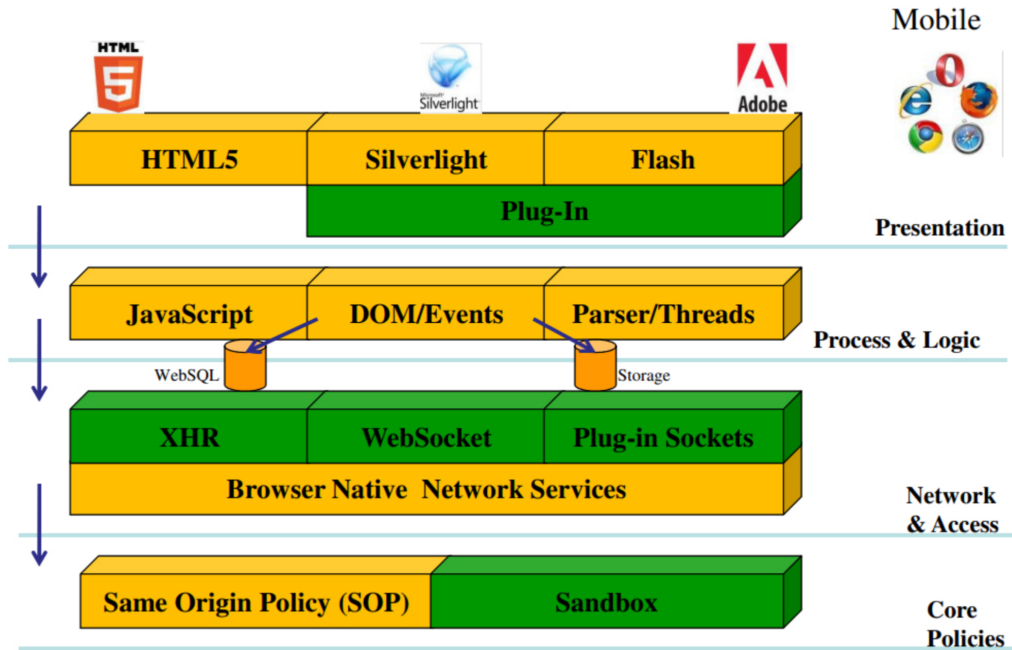


Figure 3.2: Browser architectural layers, [*OWASP* (2011)].

Core Policies. This layer defines the security mechanisms of a browser. Here, are defined mechanisms and policies responsible for ensuring the security in all the browser features. Of all security policies, there is one that takes special importance, which is the Same Origin Policy. HTML5 relaxes this policy by introducing a new policy called CORS (Cross Origin Resource Sharing).

Network & Access. This layer defines the browser connectivity services like XHR (versions 1 and 2), web sockets, plug-in specific sockets and the browser native network services. Plug-ins like Flash, Silverlight and other plug-ins have their own socket implementation. HTML5 uses web sockets and XHR (v2) requests.

Process & Logic. This is the core layer in a browser, as it defines components like Javascript, DOM, Events, Parser and Threads. HTML5 add several new features to this layer, like WebSQL, storage and enhanced DOM.

Presentation. Layer which defines the direct interaction with the end user. The most basic presentation component are the HTML tags. Also, HTML5 introduces a new set of tags which are directly integrated into the browser features, as they don't need a plug-in. Also in this layer are plug-ins like Flash, Silverlight, etc.

The HTML5 enhancements radically change the attack model for the browser. With more local storage and offline caching, the browser will contain much more

sensitive data and will make the browser an even more attractive target. As so, the standards which define the browsers policies and security, are going through a full renovation.

There are multiple attack vectors that explore the browser model. Figure 3.3, shows how different kind of threats can affect browser's layers.

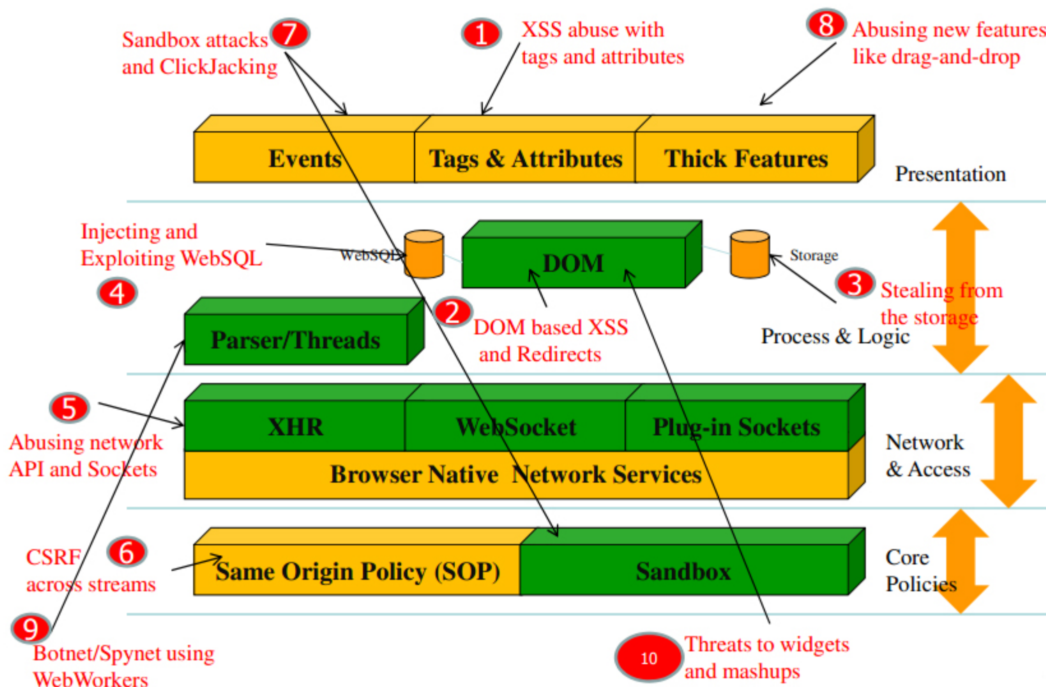


Figure 3.3: Browser attack vectors, [*OWASP* (2011)].

In Figure 3.3, ten possible threats are identified, distributed by the four layers, and all of them have one common property: they all are introduced or improved by the HTML5 specification.

During the course of this chapter, it will be described specific vulnerabilities introduced by HTML5.

3.2 Introducing API and Vulnerabilities

Figure 3.4 shows a high level diagram, which shows how HTML5 features relate to each other. This diagram briefly some the HTML5 features, and more important, explains the relationships between them from a browser context. The diagram, and

the respective relationships, are explained in the report [*Michael Schmidt* (2011)] as follows:

"DomainA.csnc.ch represents the origin of the loaded website which embeds three Iframes of different sources. The Iframe loaded from untrusted.csnc.ch is executed in a sandbox and does not have the permission to execute JavaScript code. The Iframes loaded from anydomainA.csnc.ch and anydomainB.csnc.ch are communicating to each other making use of Web Messaging. Custom scheme and content handlers are registered by domainB.csnc.ch which is requested if the user requests an appropriate resource. From domainC.csnc.ch additional resources are loaded using Cross Origin Resource Sharing. Geolcation API, Offline Web Application, Web Storage and Web Workers represent HTML5 UA features that can be used by the websites. In this example anydomainB.csnc.ch exemplarily makes use of all these features".

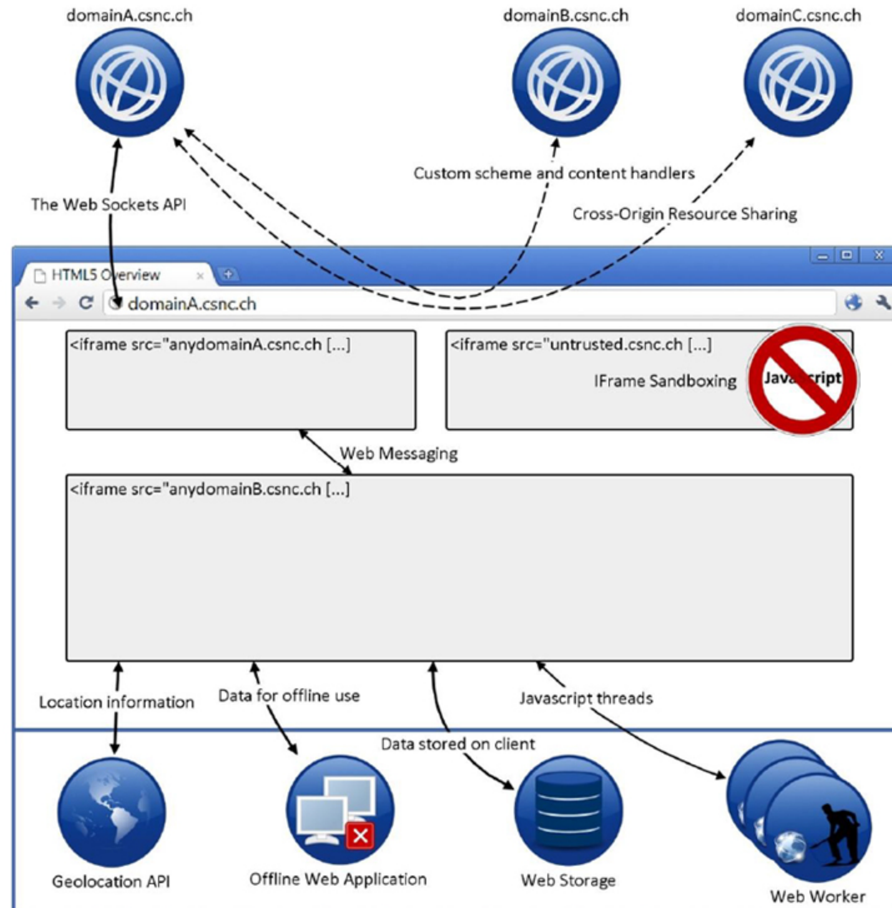


Figure 3.4: HTML5 security overview by Compass, [*Michael Schmidt* (2011)]

3.2.1 CORS - Cross Origin Resource Sharing

XMLHttpRequests is a very commonly used communication API in modern applications used with the AJAX technology. Prior to CORS, these requests were limited to communications within the same domain. CORS enables XHR communication to be made across different domains. This is probably the feature in HTML5 with the most potential to introduce security problems because it reduces the most fundamental security policy in a browser, the Same Origin Policy.

With the Same Origin Policy, a website can only communicate and access resources within their origin domain. This is a good and secure policy but it imposes some limitations, specially for web applications which are composed of several different parts displaying data from multiple origins at the same time. Usually, to bypass this limitation, developers use a proxy server compromising the efficiency of the application and introducing another point of failure to the system.

As stated before, CORS makes it possible to communicate across domains with XHR. This can be achieved by defining a new HTTP Header "Access-Control-Allow-Origin", which allows the developer to define the list of domains authorized to access response. A simple and well defined example of this header is as follows:

```
Access-Control-Allow-Origin: "http://example.um.pt"
```

Listing 1: Good implementation of a CORS header

By placing this header in an HTTP response, a developer is defining that only web applications with its origin domain from "example.um.pt" are allowed to access the response. All the other requests from foreign domains will be block and will fail when attempting to access the response.

This feature overcomes all the limitations introduced by the Same Origin Policy and brings a all new dynamic to web applications, specially the ones built out of many parts from multiple origins. It also makes implementations easier, but without the proper use by developers it can seriously compromise the security of an application.

This is not completely new, Flash has a similar mechanism which uses a file named "crossdomain.xml" to define a list of authorized domains. But also, Flash is documented as an exploitable plugin, and this file has been the route cause for some documented vulnerabilities.

In the most trivial situation we will see developers setting the header like this:

```
Access-Control-Allow-Origin: "*"
```

Listing 2: Bad implementation of a CORS header

By placing this header in an HTTP response, a developer is allowing the communication with any domain, allowing a XHR to be sent across multiple domains without asking the user for permission. This open a big range of possible attack vectors [*Michael Schmidt* (2011); *McArdle* (2011a); *OWASP* (2013b); *Kuppan* (2010); *Ryck et al.* (2011)]:

Session Hijacking. By allowing the communication with any domain, the attacker would be able to make requests on behalf of a victim, therefore abusing the user's session. This can lead the attacker into user's sensitive information, and (if it is a authenticated session) accessing any information the user has access to and perform any action the user has permission for.

Bypassing Access Control. CORS makes it even simpler to access internal websites from the internet. This can be seen as an improvement over CSRF (fifth vulnerability in the OWASP Top 10 from chapter 2). With CORS, bypassing access control is as simple as luring an internal employee to access a malicious website from the internal network and, automatically, the attacker will be able to bypass the firewall and access the internal network with a XHR.

Remote File Inclusion. RFI is an attack that leverages from pages dynamically loaded by Ajax requests, having the response loaded into the HTML of the page. For example, a page that loads the logout page based on the link:

```
http://www.example.um.pt\\index.jsp?page=logout.jsp
```

Listing 3: RFI before CORS

The contents of the logout page would be fetched, added to the current page and presented to the user. Until now, this vulnerability was limited by the Same Origin Policy. With CORS, this attack gains a all new dimension allowing attackers to perform actions like this:

```
http://www.example.um.pt/index.jsp?page=http://
                                    www.from.attack.another.domain.com/home.jsp
```

Listing 4: RFI after CORS

The given code loads the attacker's contents and embeds it in the vulnerable application, running the code in the victim's browser.

Cross site posting. This is an extension of the RFI presented above. In this case, instead of using the attack to embed code into the page, the attacker tries to redirect requests to his site and steal sensitive data. The response of the Ajax request

will probably not be processed by the requesting site but the attack will still work.

Distributed Denial of Service Attack. By placing a link with malicious script in a popular and secure website, and by luring a great number of users to click on it, an attacker will be able to overload a server with XHR requests. If the number of users clicking the link is big enough, it will be successfully launched a distributed denial of service attack against the website. This attack will be possible even with the Allow-Origin header properly defined because the requests will still be sent and processed.

Scanning internal networks. An attacker can exploit CORS ability to external communication and determine if some domain name exists or not. Even if the victim domain has the Allow-origin properly defined, this study will be possible by analyzing the response time of the requests. This will be as simple as send the request to multiple random domains and analyzing the response type and time.

Reverse Web Shells. CORS can be exploitable to set-up a browser equivalent of a reverse shell. If an attacker is able to hijack the victim's session using XSS, and inject a malicious script, this script will start to communicate with the attacker's server using a XHR and the attacker will be able to browse through the victim's affected session. The big advantage when comparing it with session hijacking is that this attack also works within applications not accessible directly for the attacker. Basically, the attacker uses the user's session and browser as a proxy. An open source tool named "Shell of the Future"[1] [*Kuppan* (2010)] was already released and implements this idea.

It would be interesting but it's not possible, in the context of this work, to get into all the security details about CORS. There are much more possible attack vectors surrounding this feature, that would lead to an extensive analysis.

The better strategy to avoid having security issues with CORS, is to properly define the list of authorized origins headers. But the browsers and the server-side applications will also have a very important role in this feature safety, because a bug in one of them compromising the origin headers, will be enough to be exploited into an attack.

3.2.2 Web Storage

Prior to HTML5, the information stored natively by the browser was limited to cookies. However, there have always been several ways for web applications to store

[1] Shell of the Future User Guide, 2013: "http://www.andlabs.org/tools/sotf/sotf.html"

data, either by using browser extensions (Webkit, Safari and Chrome have supported storage before HTML5 appears) or third-party plug-ins (like Java, Flash, Silverlight, Google Gears[2], etc.).

With the inclusion of client-side storage in the HTML5 specification, web applications will be able to store data in the client's browser and access it using simple javascript. This aims to improve web applications performance, security and to give the developer much more control over the stored data.

Client-side storage provides two different ways for storing data in the client's browser: Local Storage [*West and Pulimood* (2012), *McArdle* (2011b), *Trivero* (2008)] and Database Storage [*McArdle* (2011b), *Trivero* (2008), *Michael Schmidt* (2011)].

Session storage allows a developer to store text values in a key/value structure (which can be accessed by the key). The data expires when the browser tab is closed and is not shared between multiple tabs of the same domain, thus preventing from data leaking. With cookies, the data was shared by multiple windows of the same domain, making it hard, for example, to have two different shopping carts in two different tabs.

Much like Session Storage, Local storage only allows a developer to store text values in a key/value structure. However, Local Storage is shared between multiple tabs of the same domain and it's only restricted by the Same Origin Policy. This means that a local storage value it's only accessible for it's creator website, but it's the same within multiple tabs. Another difference in local storage is that a value never expires, and it's only deleted if explicitly deleted in code or if the user deletes the browser data.

One key advantage over the use of cookies is that, in Web storage, values are not sent to and from the server in every request, therefore reducing the network traffic and improving efficiency. Another advantage is that it increases the storage space per domain to 5MB (more, depending on the the browser's implementation). Another, but less evident difference over cookies is that a value stored in a web storage through a HTTP connection can not be accessed within a HTTPS connection and vice-versa. With cookies, this was allowed as long as both request were from the same domain.

Besides Web Storage, HTML5 introduces Database Storage which is very similar to Google Gears, and allows a web application to store structured data on the client-side using a SQLite database. This feature opens the door to the development of very powerful applications. Like in Web Storage, it only allows the web application to store strings, and the data is only accessible in the same domain.

[2]Google Gears online page: "https://developers.google.com/gears/"

Different paths in the same domain will have access to the same data. If the data needs to be different for each path, the web storage must be avoided.

Since the storage is associated with the browser and not with the user, any other user who may access the browser will have access to the stored data. As so, if the data is sensitive, developers must, at least encode the data before storing it and authenticate users before decoding it. Also, for session handling, it may be good to keep using cookies mainly because of the expiration period.

Next, it's a list of some security situations which can emerge from the use of Web Storage [*Michael Schmidt* (2011); *McArdle* (2011a); *OWASP* (2013b); *Kuppan* (2010); *Ryck et al.* (2011)]:

Session Hijacking and information stealing. If sensitive data is stored in local storage, and the website has a XSS vulnerability all the data is compromised. Therefore, if session credentials are stored in local storage the user becomes vulnerable to session hijacking. There is a method in local storage specification that makes stealing information even easier. With the key(i) method, an attacker can easily get all the local storage keys, and therefore get all the data stored.

User tracking. User tracking is already a reality using cookies. But this attack can be greatly enhanced with the use of local storage. User tracking consist and track the user through multiples sites and multiple sessions. Local storage never expires, and so can be used to store a very long history of user's actions. Local storage only stores strings, but this problem can be overcomed with a simple string concatenation.

Vulnerable to malware. A web application might be totally secure and free of any kind of vulnerabilities. But if the user's machine is compromised by a malware, it will be able to scan the browser's data stored in the disk.

Stored XSS attacks. If the web application load data or code from local storage without proper sanitization, local storage could also become a powerfull tool to inject malicious scripts into web applications and perform Cross-Site Script attacks.

Client-Side SQL Injection. SQL Injection have always been considered a server-side problem, but already with Google Gears, it became also a client-side problem. Like, in server-side SQL Injection this problem happens when parametrized queries are not used.

Local resource exhausting. WebSQL databases are restricted to 5MB in size. When an application tries to store more than 5MB some browsers ask user for permission but another ones simply don't care. This can be used by an attacker to fill the victim's hard disk with spam.

DNS spoofing/cache poisoning. This is a well know vulnerability one cannot assume that a host who claims to belong to a certain domain, really is from that domain. As so, TLS must be used.

3.2.3 Offline Web Application

Another feature introduced by HTML5 it's called Offline Web Application. This feature works with the use of an application cache designed to enable offline web browsing.

Any site can create offline versions of itself in the browser cache, by defining a set of files which are needed for the application to work offline. To do this, the developer must add to a manifest file a list of resources (CSS, HTML, JS, etc). This manifest file will be parsed by the browser, which will download and keep updated all the resources listed. Next time the user tries to access the website without an Internet connection, the browser automatically swaps to the local copy.

The manifest file can be named and located anywhere on the server. Then, it must be added to the attribute manifest in the html tag:

```
<html manifest="/cache.manifest">
```

Listing 5: Manifest attribute

In the HTML5 specification there are no major security implementation details about this feature. However, it is stated that end users must be advised to clear the browser's cache every time they access the Internet from an insecure network. Also, end users must be warned to only accept offline caching from trusted websites.

Some possible security problems introduced by this feature are now described [*Michael Schmidt* (2011); *McArdle* (2011a); *OWASP* (2013b); *Kuppan* (2010); *Ryck et al.* (2011)]:

Cache poisoning. Cache poisoning attacks was already a threat before HTML5, but now it gains even more relevance. By poisoning the browser's cache, an attacker can inject scripts and have them alive for longer durations. The scripts can be used to perform tasks like steal user's credentials or even to keep an eye on a particular domain.

User tracking. The data stored in the Offline Web Application cache stays on the client's browser until the server sends an update or the user deletes the cache manually. As with Web Storage, the data stored offline can be used for user tracking.

3.2.4 Web Messaging

In HTML4, documents in different domains are not allowed to interact with each other. This impose a huge limitation over those applications composed from multiple embedded webpages and (more recently) javascript widgets. Until now, developers are used to have two main methods to integrate external resources: using iframes or inline javascript.

Using iframes (mainly used in the inclusion of embedded webpages) is secure, but the contents loaded inside the iframe are isolated from the context of the main application and cannot access the DOM.

With inline javascript (mainly used in the integration of javascript widgets), the code is directly included into the main page and have complete access to the context and the DOM. This is very useful but introduces serious security issues.

HTML5 specification introduces Cross Document Messaging, which allows the communication between pages from different domains, in a way designed to prevent cross-site scripting attacks. With this feature, it will be possible to include external resources into embedded iframes and the iframes will be able to communicate with each other and with the embedding web application.

Web messaging also integrates another way of communication called Channel Messaging, which enables independent pieces of code, even if running in different contexts, to communicate directly. The communication is implemented as two-ways pipes, with a port on each end and messages delivered asynchronously as DOM events.

In terms of security, integrating external resources with web messaging it's a big security improvement. However, it can also introduce security risks if not implemented with caution. Some possible security threats are now presented [*Michael Schmidt* (2011); *McArdle* (2011a); *OWASP* (2013b); *Kuppan* (2010); *Ryck et al.* (2011)]:

Injection of malicious scripts. Attackers can leverage from the communication between documents, to send malicious content or steal sensitive data. Developers must always check the origin of the message, ensuring that the message is from the domain which it is supposed to be. Also, even with a proper checking of the origin attribute, developers must validate the message and ensure that the data received is in the format is supposed to. Finally, the data received must never be used directly as innerHTML or passed into the javascript function "eval()".

Disclosure of sensitive data. In the most trivial flaw, we will see developers using the "*" in the Target attribute of messages with sensitive information. By using "*", there is no guarantee that the message will be delivered only to the intended

recipient. This can lead to situations where the message is delivered to the wrong recipient, disclosing sensitive data.

Denial of Service Attacks. Developers who accept messages from any origin may be vulnerable to a denial-of-service attack. As it accepts messages from any domain, an attacker may be able to send a high volume of messages per minute, leading to expensive computations and network traffic. HTML5 specification encouraged developers to employ rate limiting (only accepting a certain number of messages per minute) to avoid such attacks.

3.2.5 Web Sockets

Web Sockets is an API which allows a bidirectional, full-duplex communication over a TCP socket. TCP Sockets are already known and very used in the IT world, and are now being applied to the Client/Server communication in a web application. Like CORS, Web Sockets connections can be established across domains.

Web Sockets will win special relevance in the development of real-time interactive applications like chats, games, etc. Before Web Sockets, AJAX pooling mechanisms were used to simulate a TCP connection, but have been proved as an inefficient solution.

However, Web Sockets API doesn't handle authentication and authorization. If sensitive data are to be transfered, authentication and authorization must be handled by the application. Also, in every packaged transfered, origin and destination must be confirmed, and the contents of the packaged must be validated to avoid malicious inputs. Furthermore, Web Sockets over SSH (wss://) must always be used to encrypt the messages. If not, ("ws://") messages will be transfered in plain text and a Men-In-The-Middle attack will lead to the disclosure of the data.

Like any other feature, Web Sockets can introduce security problems [*Michael Schmidt* (2011); *McArdle* (2011a); *OWASP* (2013b); *Kuppan* (2010); *Ryck et al.* (2011)]:

Port Scanning. Web Sockets can be used for port scanning of internal networks. An attacker can have a malicious page, which the only goal is to open sockets and do a port scanning on internal IP addresses. If the scanning results in an interesting open port on the internal network, a tunnel can be established with the victim's browser. The attacker will, then, be able to bypass the firewall and access the internal network. And all the victim did was accessing the attacker's malicious webpage.

Access the connection. In applications using Web Sockets, XSS vulnerabilities gain even more relevance, as the attacker will have access to the existing connections

and will even be able to create new ones. By accessing an existing connection in the victim's page, an attacker will have the same permissions the user have over that connection. So, he will be allowed to perform actions like log all the messages sent and received by the victim, send messages in behalf of the victim, etc.

Remote Shell. Another possible scenario is the attacker to create a remote Shell. If an application is vulnerable to XSS attacks, and if the attacker is able to gain access to the victim's page, he will be able to establish a Web Socket connection. Then, he can use that connection to execute any JavaScript he wants on the victim's page. The connection will stay open until the victim close the browser page.

Distributed Denial of Service Attack. Like in CORS, an attacker can use the victim's vulnerable page to perform distributed denial of service attacks. By placing a link with malicious script in a popular and secure website, and by luring a great number of users to click on it, an attacker will be able to overload a server with Web Socket connections and requests.

3.2.6 New Tags

HTML5 will support new elements and attributes. This expands the attack surface for code injection attacks. Every new tag which allows Javascript execution, represents a new possible attack vector for XSS and CSRF. Even applications which are not vulnerable to Javascript, may become vulnerable because existing XSS filters may fail against new and unknown tags.

Exploiting autofocus. In HTML5, a new ”autofocus” attribute is introduced, which allows an element to be automatically focused. Before HTML5, injected javascript will be placed in event handlers like ”onmouseover” or ”onclick”, which requires the user interaction to be triggered. With this new attribute, the javascript can be placed in the ”onfocus” event handler, and by also setting the ”autofocus” attribute, the script will be triggered automatically, without the need of any user interaction. This makes it much easier to execute injected scripts.

Form tampering. HTML5 also adds new attributes to the submit button, which overrides the attribute values defined in the form. New attributes are ”formaction”, ”formenctype”, ”formmethod”, ”formnovalidate” and ”formtarget”. Also, a new ”form” attribute is available to specify the form that an element belongs to. Now, one element can be outside of the ”form” node and still belong to that form, if indicated with the attribute ”form”.

So, if a page is vulnerable to a XSS attack, an attacker will be able to alter how

the forms on that page behave. Due to the newly-provided capability to introduce elements outside a form and override form properties, the attacker will just need to introduce a new submit button in the page and use the new attributes to redefine the form behavior. Also, the button can be placed anywhere in the page and associated with the desired form with the "form" attribute. So, if the attacker can lure the victim into click in the new submit button, he will successfully be able to steal the form information, by submitting the data into a destination of his choice.

3.2.7 Web Workers

The main goal of web Workers is to run background JavaScript so that long tasks can be executed without making the page slower or unresponsive. Currently, JavaScript has one big limitation: all its execution it's done in one thread. Workers overcome this limitation, by allowing the developer to create multiple JavaScript threads that will run independently from each other. This will allow powerful scripts to be executed in the user computer without him noticing it and will be very useful when it comes to resource-intensive applications.

Workers don't have access to the DOM of the calling page, and all the communication must be done by message passing. Workers are also allowed to use XHR to perform in-domain and CORS requests. In the specification is stated that, generally, workers are expected to be long-lived, have a high start-up performance cost, and a high per-instance memory cost.

Web workers may not introduce security vulnerabilities directly, but they can introduce a way of an attacker run code in background without the user notice it. This is danger and can introduce situations like this [*Michael Schmidt* (2011); *McArdle* (2011a); *OWASP* (2013b); *Kuppan* (2010); *Ryck et al.* (2011)]:

Spamming. Although workers do not have access to the DOM, they still can be used to perform XHR requests and consuming CPU and memory resources. As so, they can be used to spam and overload the user browser and computer. It's very important not to create Workers directly from user supplied input. However, if the application is vulnerable to XSS attacks, an attacker may still be able to have Workers executed in the user browser.

Distributed Denial of Service. It was already described in CORS and Web Workers definitions, the possibility of achieving a DDoS attack. However, with Workers this attack wins a all new dimension. A Worker can be executing in a user computer, sending thousands of XHR requests per second and the user may not even

notice. In [*Michael Schmidt* (2011)] is stated that, by combining CORS and Workers is possible to send, with only one browser, about 10000 requests per second to a server.

If an attacker can lure a great number of users all over the world to invoke the malicious script, and with each user performing 10000 requests per second, this would represent a huge server overload.

Distributed Brute Force Attack. Brute force attacks requires a lot of CPU processing to be performed. Much like in the DDoS attack described above, an attacker can lure a great number of users to invoke a malicious script and use Workers in the victims browsers to distribute the CPU load.

Web Workers and Web Messaging. All the messages exchanged with the Worker have to be validated and escaped in order to avoid XSS vulnerabilities. Also, the messages received must never be used directly as innerHTML or passed into the javascript function "eval()".

Furthermore, if an application already has XSS vulnerabilities, an attacker may be able to inject a stealth Worker which will allow the attacker to monitor changes in a particular DOM. Despite the worker inability to access the DOM, the attacker can use event listeners to send informations to the worker.

3.2.8 Sandbox

HTML5 introduces a "sandbox" attribute on iframes. By defining the sandbox attribute, the following resources are disabled: plugins, forms, scripts, links to other browsing contexts and the same-origin treatment of the content hosted by the iframe and the content which hosts the iframe.

However, these restrictions can be relaxed and certain features can be re-enabled by defining sandbox attribute value: allow-forms (re-enables forms), allow-scripts (re-enables scripts), allow-top-navigation (re-enables links), allow-same-origin (re-enables same-origin treatment).

This feature as already introduced one security vulnerability:

Bypass javascript security. Web framing attacks such as clickjacking use iframes to hijack a user's web session. The most used defense method, is called frame busting, and prevents a site from functioning when loaded inside an frame. The original version of frame busting can be easily bypassed using sandbox. Setting the sandbox attribute disables JavaScript within the iframe and since the frame busting relies on Javascript, it will not have the desired effect.

Very recently, an update over the original frame busting method was released that prevents this to happen. However, a big number of developers may not be aware of the need to update their code, leave a great number of applications in risk of clickjacking attacks.

3.2.9 Drag and Drop

This feature allows the user to drag and drop content between pages in the browser, from the browser to the computer or from the computer to the browser.

Drag and Drop operations dealing with sensitive data must be dealt with extreme caution. For example, developers must only consider a drop to be successful if the user specifically ended the drag operation. In any other situation the operation must be canceled and the drop must not be completed, therefore not compromising the data.

Text-field injection. Drad and Drop feature can be used to inject contents into an application simply by luring the victim to perform a Drag and Drop action. If the application does not ensures that only safe content is dragged, malicious scripts can be dropped into the victim page, therefore performing a XSS attack. This is not new, it just introduces a new way of luring victims, as the attackers can even camouflage the attack as a game that requires the player to drag and drop items.

3.2.10 Geolocation

The Geolocation API provides a high-level interface to get the user's physical location based on the GPS coordinates, latitude and longitude. As other features, before Geolocation, getting the users position was only possible with external plug-ins.

Geolocation uses different sources to get the coordinates (e.g. GPS, WiFi and mobile phone networks, IP address, used configured location) and different sources mean different accuracies and response times, but they all get the job done.

Geolocation can raise privacy issues. The application always has to ask the user for permission before accessing his location information (however, if this decision is remembered or not, depends on the browser implementation). This may seem a good security policy, but, this kind of security measure is not always as efficient as it should be. Users not always understand the security risks and can be lured into accept the request. As so, is very important to make users aware of the risks and advise them

to only share their location to trusted service providers.

Possible threats introduced by the Geolocation feature will include [*Michael Schmidt* (2011); *McArdle* (2011a); *OWASP* (2013b); *Kuppan* (2010); *Ryck et al.* (2011)]:

Tracking victims. Once the attacker gets the user permission, it has access to the user's current location. Also, the attacker can keep record of the various accesses to the application and track the user movements in real-time.

Disclosure of Personal Information. With some precision, the location information can be used to identify the user's home address, places that he often visits, etc. This kind of information opens a all new range of attack vectors outside of the computer world.

Breaking Anonymity. If malicious web applications share the location information of their users, this information can be used to identify users between multiple domains. If the user is registered in one of the applications, the user's location information can be used to identify the same user in another application. Of course, this will not have 100% of efficiency because information may not be accurate enough or because there may be multiple users with similar locations. However, this is still a valid threat which can compromise the anonymity of the user.

3.3 Summary

During this chapter, each one of the new HTML5 features were described. A good conclusion comes out of this chapter: HTML5 was, from the beginning, designed with security in consideration. However, if the developer doesn't follow the security considerations presented in the specification, HTML5 can introduce some new and dangerous attacking vectors.

HTML5 will give the attackers some new ways of attack, which can be a useful weapon for exploiting already vulnerable web applications. However, HTML5 will not compromise secure web applications, and there is no need to redefine strategies for the existing web applications.

One of the goals of this dissertation was to study the new features of HTML5, and understand if they can introduce new security vulnerabilities. In general, HTML5 will not introduce new vulnerabilities. However, it will boost some of the existing ones, mainly Cross-Site Scripting. This comes from the inevitable consequence of bringing more data and operations from the server to the client side.

At this point, it were already studied the vulnerabilities present in web applica-

tions, and the consequences of integrating the new HTML5. In the next chapter, it will approached another goal for this dissertation: study the process of building a static analysis tool and the existing open source tools.

CHAPTER IV

Static Analysis: Background

Manually reviewing the security of a web application is a long and complex process. In fact, a complete manual review over a modern web application can be a very time-consuming and expensive process. Therefore, automated tools are more and more used. There are tools designed only to help the analyst in his review, while others are designed to be more independent and to perform full analysis, generating reports for the analyst to work upon.

Automated approaches can be further classified into two main categories: static analysis or dynamic runtime approaches. In this chapter these two categories will be described in an attempt to clearly differentiate them and highlight their pros and cons.

Then, it will be studied the process of building a static analysis tool, addressing all the stages of development, as the strategies and data structures needed during the process. Dynamic tools will not be addressed here, as they fall out of the context of this work.

Finally, a small set of open source static analysis tools will be studied, describing their strategies, advantages and limitations. Not all of the tools described target security, but they all fit into the profile of interest in the context of this work.

4.1 Reviewing security

4.1.1 Dynamic Runtime Approaches

Web applications are constantly changing, the source code is not always available, and new changes can represent new security vulnerabilities. In these situations, dynamic approaches are very useful because they detect flaws while executing the

audited application, after deploy and without the need for the source code.

A vulnerability detected by a dynamic tool, normally means that the tool was able to successfully prove the existence of that vulnerability, leading to less false positives and more reliable results. This reduces the human time needed to review the results.

Another advantage of dynamic approaches is that they are not language-dependent. Many dynamic strategies and mechanisms are independent from the language, and can be applied against any application.

However, automated approaches can also give a false sense of security that all vulnerabilities were found and that everything is tested, while that is not true. Dynamic approaches don't test all the possible execution paths, only static analysis approaches can do that. Instead, only the paths invoked during the execution are tested. Also, it's more difficult to trace the vulnerability back to the exact location in the code, taking longer to fix the problem.

4.1.2 Static Analysis Approaches

Static analysis (also referred as code analysis) has been a topic of strong research for more than 30 years [*Chess and West* (2007); *Pistoia et al.* (2007); *Dagenais and Hendren* (2008)] and is still actual.

Static analysis can give developers precise and detailed information about their entire application. Any tool which analyzes code without executing it, is performing static analysis. Also, static analysis may not always be made over the source code. There are tools which perform the analysis over different compilation levels. In Java, for example, there are many tools which perform the analysis over bytecode. A good example is FindBugs [*Ayewah et al.* (2008)], presented later in this chapter.

In the scope of this work, static analysis will be approached as a method to perform security analysis over software applications.

Security is sometimes overlooked in the software development lifecycle. This is a huge error, developers should have well defined strategies and methodologies to approach security. Developers need to examine and review the source code, multiple times, to identify possible security flaws. Thats when static analysis tools step in. Static analysis tools are helpful during multiple stages of the software development lifecycle. This is a great advantage over dynamic approaches, allowing flaws to be found earlier and therefore, reducing the effort and the costs to fix it.

When compared against runtime approaches, static analysis tools find more flaws and have no runtime performance overhead because they are done before deploy. However, they produce more false positives which reinforces the need for a human

component. All reported errors must be manually examined in order to identify possible false positives and find the true vulnerabilities in the code.

Another advantage of static analysis is that results can point out to the exact location where the problem was detected, therefore simplifying the process of solving it. Also, static analysis are capable of examining all the possible execution paths in a program, and not just those invoked during execution. This means that static analysis tools may find vulnerabilities that runtime testing tools won't.

Static analysis tools have been in constant changing in the last decade, and have evolved to became more efficient and more capable. In [*Lopes et al.* (2009)], the first static analysis tools are described as "difficult to use, tedious, and limited in their ability to find real bugs (they could only find a list of functions known to be dangerous)", while more recent tools are described as capable of "parse almost all the languages for many common coding problems within different categories, including complex error patterns". The authors of [*Lopes et al.* (2009)] justify this evolution with the complex techniques which have been added to static analysis tools over the time (giving as an example metrics analysis, context analysis, semantic analysis and the use of metadata to define rules and inner-function dependencies).

When testing such tools, the evolution becomes evident. The tool Lint was the original static code analyzer of C code, and represents the ground zero of static analyzers. There are, still, tools based on Lint, usually known as Lint-based. Are examples of lint-based tools: PC-lint[1], FlexeLint [2], JSHint[3] or Splint[4].

There are a big number of static code analysis tools available today, ranging from the most trivial syntax parsers to more complex tools combining multiple methods and strategies. Later in this chapter, it will be described and compared a series of static analysis tools.

4.1.3 Hybrid approaches

Theoretically, an application security review should be a two-step process where the application is submitted through independent code analysis and runtime tests. Thus, static analysis can be used to find earlier vulnerabilities and fix them before deploy the application. More ahead, runtime tests can be made to confirm the static analysis results, and to make the tests even after the application goes into production.

[1]PC-lint online page, 2013: "http://www.gimpel.com/html/pcl.htm"
[2]FlexeLint online page, 2013: "http://www.gimpel.com/html/flex.htm"
[3]JSHint online page, 2013: "http://www.jshint.com/"
[4]Splint online page, 2013: "http://www.splint.org/"

JNuke [*Artho and Biere* (2005)] is a great example of a hybrid solution. JNuke is a framework for fault detection, which combines static and dynamic analysis. There are two main advantages with this approach: static results can be applied in runtime validation for verification and, by using static information, the runtime overhead it's reduced.

JNuke is presented by its authors as follows: "By using a graph-free analysis, static analysis remains close enough to program execution such that the algorithmic part can be re-used for dynamic analysis. The environment encapsulates the differences between these two scenarios, making evaluation of the generic algorithm completely transparent to its environment. This way, the entire analysis logics and data structures can be re-used, allowing for comparing the two technologies with respect to precision and efficiency".

DyTa [*Ge et al.* (2011)] it's another project that combines both static verification and dynamic test generation. Again, the strategy is running dynamic tests over static results to improve efficiency. DyTa is presented by its authors as follows: "The static phase detects potential defects with a static checker; the dynamic phase generates test inputs through dynamic symbolic execution to confirm these potential defects. DyTa reduces the number of false positives compared to static verification and performs more efficiently compared to dynamic test generation".

DyTa and JNuke are two examples, from a big range of academic and non-academic studies, of combining static and dynamic approaches. But again, the focus of this work will be in the static analysis perspective.

4.2 Building a Static Analysis Tool

Static analysis tools play an essential role in the security of web applications, helping both developers and security reviewers. Developers can use them, during the development stage, to fix bugs and security flaws as the application is being built. Security reviewers can use them as a tool of assistant to find possible flaws and then, manually analyze each possible flaw and evaluate if it is significant or not.

The process of building static analysis tools inherits a lot from the compilers world [*Aho et al.* (1986)]. In fact, all static analysis tools have a similar way of working. As we can see of Figure 4.1, at a very high level, they all receive source code files, build a series of Intermediate Representation (IR) models and perform the analysis on top of those models. The output of the program reflects the results of the analysis made.

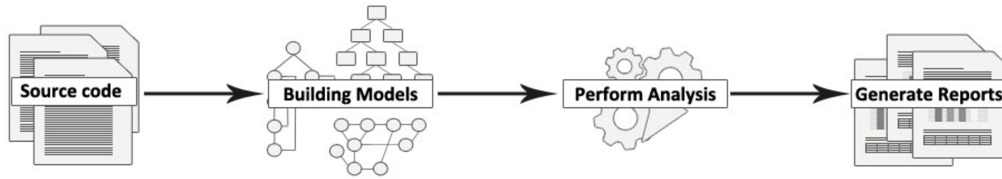


Figure 4.1: High level model of a static analysis tool.

Next, each of the stages from Figure 4.1 will be described in more detail.

4.2.1 Source code

The main advantage of a static analysis tool is the fact that it's executed prior to deploy. So, the analysis is applied over the source code of the application. Some applications support multiple languages, while others are more specific and target only one language. In fact, there are some tools even more specific that target only a specific context of the language.

By having access to the source code, static analysis tools are able to get a precise intermediate representation of the code (IR) and find flaws in the application that are impossible to find with dynamic testing tools. But, in some situations, where the source code is either not available or incomplete, it may be difficult to build a complete and precise intermediate representation (IR) of the program.

To get around this, reverse engineer techniques can be applied. There are also some static analysis related projects concentrated on reverse engineering incomplete or compiled code, aiming to diminish the dependence of static analysis techniques from the source code. A very good example is the framework described in [*Dagenais and Hendren* (2008)] that, as described by the authors, is "a framework that performs partial type inference and uses heuristics to recover the declared type of expressions and resolve ambiguities in partial Java programs". The framework "produces a complete and typed IR suitable for further static analysis".

This is very interesting because, in situations where the source code is compiled or incomplete, frameworks like this can recover important contents of the application, making it suitable for static analysis. These kind of reverse engineering techniques, even if related to static analysis, represent a broad and completely different subject, that will not be further studied as it falls out of the context of this work.

4.2.2 Building models

Assuming the source code is already available, the first step when performing static analysis is to transform the source code into one ore more intermediate representation (IR) models. In the particular case of developing a tool targeting security, these IR models are normally the following:

- Abstract syntax tree (AST)
- Control flow graph (CFG)
- Call graph (CG)

In essence, building these IRs structures is abstracting particular aspects of the application sinto a set of models suited for analysis.

The following example, from [*Chess and West* (2007)], shows the process of building an AST and was developed with the ANTLR tool, ANother Tool for Language Recognition [*Parr* (2007)].

Building an AST, starts with a lexical analysis [*Aho et al.* (1986)], transforming the source code into a series of tokens. These tokens are identified by a series of rules, normally based on regular expressions.

Taking the following example:

```
if (x==1)
  y = y + 1;
```

Listing 6: Code example.

And applying the following lexical rules:

```
IF         :     'if';
OPEN_PAR   :     '(';
CLOSE_PAR  :     ')';
EQUAL      :     '=';
SC         :     ';';
COMP_OP    :     ('==' | '>' | '<' | '!=') ;
SUM_OP     :     '+' ;
TXT_VALUE  :     ( '"' (~'"')* '"' | '\'' (~'"')* '\'' );
INT_VALUE  :     ('0'..'9')+;
NAME       :     ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9')*;
WS         :     (' ' | '\n' | '\t' | '\r' )+ { $channel=HIDDEN; };
```

Listing 7: Example of lexical rules.

Then, as a result we will have the following sequence of tokens:

```
IF OPEN_PAR NAME COMP_OP INT_VALUE CLOSE_PAR NAME EQUAL NAME SUM_OP
    INT_VALUE;
```

Listing 8: Resultant sequence of tokens.

On a side note, there are some simple security tools which doesn't need to go any further. These tools work by looking for the names of dangerous functions (known to be responsible to introduce vulnerabilities) in the sequence of tokens. If the function is invoked, a warning is issued. Example of known tools which use this approach are ITS4[1] or Flawfinder[2].

Continuing with the example, the next step is to match the sequence of tokens against a set of production rules which describes a context-free-grammar (CFG). This is the parsing stage:

```
code          :     if_cond | assign_stmt ;
if_cond       :     IF OPEN_PAR comp CLOSE_PAR code ;
assign_stmt   :     var EQUAL expr1 SC;
comp          :     expr1 COMP_OP expr1;
expr1         :     sum_oper | var | value ;
expr2         :     var | value ;
sum_oper      :     expr2 SUM_OP expr2;
var           :     NAME;
value         :     INT_VALUE | TXT_VALUE;
```

Listing 9: Example of a grammar language.

By matching the sequence of tokens against the production rules, the parser will be able to derive the parse tree (or concrete syntax tree) shown in Figure 4.2 (for simplicity reasons, symbols that do not carry names were removed). The parse tree is a data structure which contains the most direct representation of the code, exactly as the programmer wrote it. Performing analysis over a parse tree is very inefficient, as it contains symbols that exists only for the purpose of making the parsing stage easier and non-ambiguous. So, the static analysis is not performed over one parse tree, but over one Abstract Syntax Tree (AST).

One AST provides a more general and standardized version of the program, removing all the symbols not needed for analysis. The AST is normally built by associating tree construction code with the more generic grammars production rules.
The AST for the example in study is shown on Figure 4.3.

[1]ITS4 online page, 2013: "http://www.cigital.com/its4/"
[2]Flawfinder online page, 2013: "http://www.dwheeler.com/flawfinder/

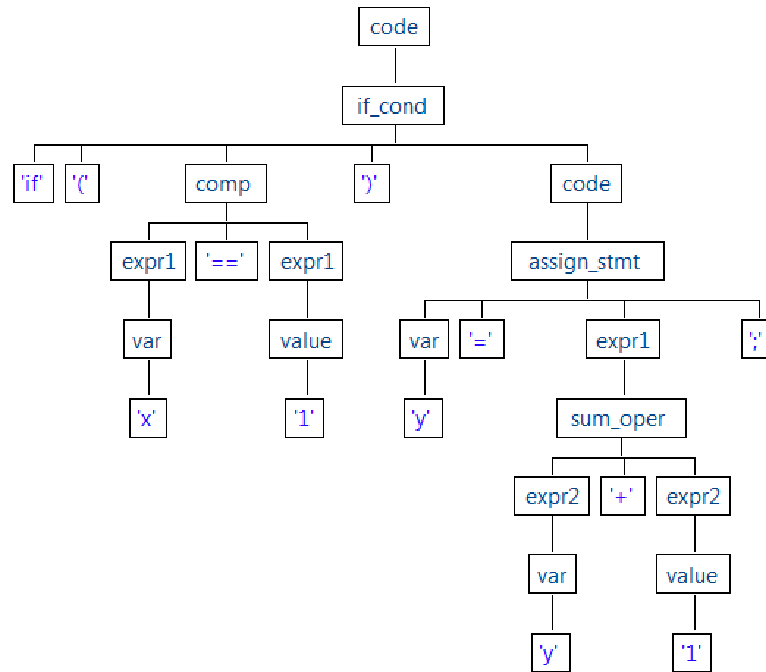


Figure 4.2: Parse tree generated in the example.

In this stage, along with the AST, it's performed the so called **semantic analysis**. Semantic analysis it's where are given meaning to the symbols found in the program. Are examples of semantic analysis: symbol resolution (building a symbol table which maps each identifier with the information related to its declaration or definition), type checking (checking for type errors), object binding (associating variables and functions with their definitions), definite assignment (requiring all local variables to be initialized before use), etc.

Until the AST construction, the process is the same for both compilers and static analysis tools (including the ones which that target security). However, in semantic analysis, each solution may have its own implementation. Semantic analysis depends directly from the purpose of the application, and different static analysis tools require different semantic analysis techniques. There are, in fact, tools which don't need to go any further, as semantic analysis is the type of analysis they are looking for (this is not the case on most of the security analysis).

At this point, and based on the AST already available, multiple options of analysis are available. As stated before, the AST provides a standardized version of the program, and it will be the basis for all the analysis made. There are multiple analysis techniques, ranging from the most primitive to the more complex and sophisticated. In fact, there are so many academics and non-academics studies, tools and prototypes

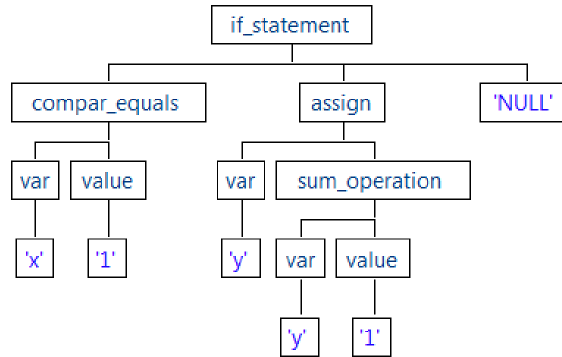


Figure 4.3: Abstract syntax tree for the example.

that it would be impossible to cover them all in one document.

There are two main techniques which can be highlighted, as they fit better into the purpose of this work:

- Pattern Based Analysis
- Data Flow Analysis

4.2.2.1 Pattern Based Analysis

Pattern based analysis is the process of parsing the source code looking for patterns. To security purposes, these patterns normally represent pieces of code known to introduce security vulnerabilities.

Pattern Based Analysis is a flow-insensitive technique, and is seen more as a solution for prevention instead of detection. It's commonly used, for example, in IDEs and syntax checkers. However, this does not mean that pattern based analysis don't fit into security analysis. It is very useful to build systems with analysis oriented by security rules (or patterns).

A static analysis tool, sometimes provides the user with the option of building their own custom security rules. Pattern based analysis it's also very useful in these situations.

Pattern based analysis can be performed in multiple stages and over different IR data structures. It can be done over the series of tokens generated in the lexical analysis. It can also be done over the parse tree or the abstract syntax tree. More rudimentary solutions can even be done over the source code.

4.2.2.2 Data flow analysis

A major challenge associated with web applications is that their most critical vulnerabilities are often the result of insecure information flow. The fact is, analyzing how data propagates throw an application is the only way to detect some type of vulnerabilities.

Data flow analysis [*Nilsson-Nyman et al.* (2009); *Khedker et al.* (2009); *Pistoia et al.* (2007); *Chess and West* (2007)] is the process of collecting information about the use, definition and dependencies of data in programs. In [*Pistoia et al.* (2007)], data flow analysis is described as "a process for collecting run-time information about data in programs without actually executing them. Data flow analysis however does not make use of semantics of operators. The semantics of the program language syntax is, however, implicit captured in the rules of the algorithms".

The first step to perform data flow analysis, is to built a **Control Flow Graph (CFG)** on top of the AST (and other IR models available). A CFG represent all the possible execution paths of a program.

Each node in the graph represents a **basic block**, which is a sequence of consecutive instructions such that control flow enters the block in the beginning and only leaves at the end, with all the instructions in the block executed. In other words, all the instructions in the basic block must be executed and there is no possibility of any instructions being skipped.

Directed edges are used to represent jumps in the control flow. If a block has no incoming edge it is called an entry block, and if a block has no outgoing edge, it is called an exit block. Back edges represent potential loops.

Data flow analysis is performed without (actually) executing the program. Therefore is not possible to determine what will be the path (in the control flow graph) that will be taken during the execution of the program. So, all the possible execution paths are simulated. A good example is the if-then-else condition: as it is not possible to know what branch of the condition the program will execute, both branches of the statement will be considered to be actual execution paths.

A **control flow path** is a path in the control flow graph that starts at an entry block (first instruction) and ends at an exit block (last instruction). The number of control flow paths in a program can range from one (very uncommon, only seen in the most basic programs) to infinite (because of the unpredictability of some loop bounds). One major task of the worst case execution time analysis is to find the longest possible control flow path.

The following example, from [*Chess and West* (2007)], is a very simple example

of a control flow graph. The code fragment shows a simple if-then-else condition followed with a return statement. The if branch has only one condition while the else branch has two conditions.

To simplify the example, basic blocks show source code. In a static analysis tool, this blocks don't contain code but pointers to AST nodes (or other IR models used). In the corresponding control flow graph, also shown in figure 4.4, it's possible to see that there are two possible control flow paths: [bb0, bb1, bb3] and [bb0, bb2, bb3].

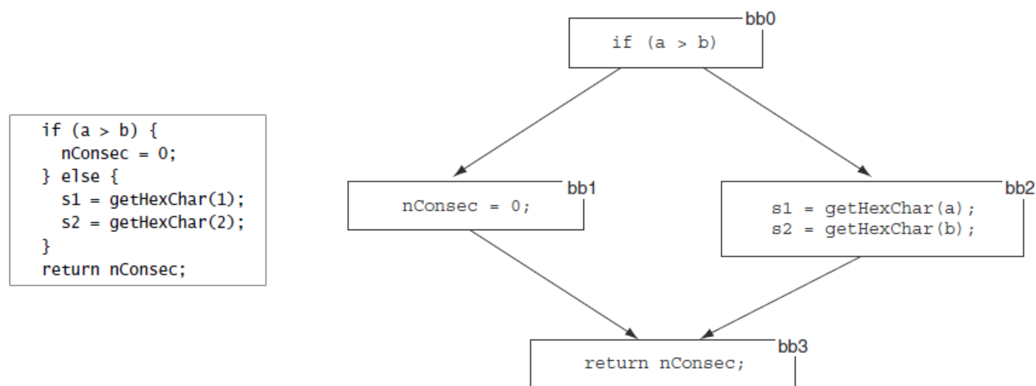


Figure 4.4: Example of a simple control flow graph.

All methods presented so far are considered part of an intra-procedural analysis. When bugs or vulnerabilities are limited to a function scope, an intra-procedural analysis is probably enough. But, when a more comprehensive analysis is needed, it may require inter-procedural analysis techniques, normally seen only in more advanced tools.

For this, at least one data structure is needed: a **call graph**, which represents the possible interaction between functions and methods in an application. In a CG, nodes represent functions and edges represent invocations between those functions.

Figure 4.5, from [*Chess and West* (2007)], shows a simple example of a call graph, and the corresponding source code. The call graph can be easily read like:

- function larry can invoke moe or curly;
- function moe can invoke curly or moe again (recursively);
- function curly don't invoke any functions.

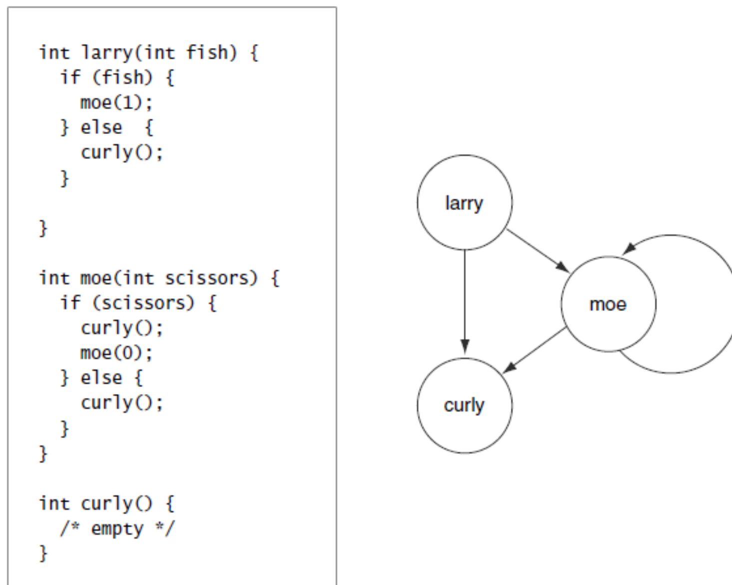


Figure 4.5: Example of a call graph.

At this point it was already explained what is Data Flow Analysis and what data structures are needed to implement it. Of course, there are also a possibility of other structures being needed, which can range from the most basic to more complex ones.

4.2.3 Perform Analysis

At this point, it's important to understand what are the possible strategies and algorithms which can be applied to approach data flow problems. Analysis algorithms work over the data structures, combining and manipulating the data in order to get the analysis information.

In this work, it will be approached three main algorithms used to perform data flow analysis. These algorithms alone are sufficient to identify many data flow vulnerabilities and are the basis for many known tools and more advanced techniques:

- Single Statement Assessment;
- Taint Analysis;
- Pointer Analysis.

4.2.4 Single Statement Assessment

SSA consists on creating new variables every time a value is assigned to a variable. This means that, in SSA, a variable can be assigned only once. In a simple case, where

a variable (var) is assigned three times in a program, after applying SSA, the program will refer to variables as var1, var2, var3.

This can be very useful to identify dataflow problems, as it becomes very easy to understand from where a variable comes.

4.2.5 Taint Analysis

Taint analysis [*Sridharan et al.* (2011); *Chess and West* (2007); *Nilsson-Nyman et al.* (2009)] is a technique which emerged from PERL, and many analysis have appeared in the last few years as taint-based approaches [*Mui and Frankl* (2011); *Haldar et al.* (2005); *Sridharan et al.* (2011)]. The goal in taint analysis is to detect the flow of untrusted information into sensitive areas, which can compromise the security of the application.

In [*Sridharan et al.* (2011)], Taint analysis is described as "an information-flow analysis that automatically detects flows of untrusted data into security-sensitive computations (integrity violations) or flows of private data into computations that expose information to public observers".

The first step to understand taint analysis is defining its terminology: A tainted object propagation problem consists of a set of source, sink and derivation descriptors.

- **Source descriptors** specify ways in which user-provided data can enter the program.
- **Sink descriptors** specify unsafe places where data may be used in the program.
- **Derivation descriptors** specify how data propagates between objects in the program.

Is considered tainted, all data possibly coming from malicious input (sources), not properly escaped, and capable of cause security flaws in sensitive points of the program (sinks). Functions which have the goal of escaping input are called sanitizers.

4.2.6 Pointer Analysis

Finding objects which a given variable or object field may point to it's called Pointer Analysis or Points-to Analysis. Pointer analysis is a well-known program analysis, which provides information needed in data flow analysis.

There is a wide variety of points-to analyses, with different degrees of precision and cost.

Pointer analysis techniques are very useful and very important when performing any kind of static analysis and security analysis are not an exception. Pointer analysis is also a know and relatively well understood as a whole independent program analysis.

4.2.7 Generating reports

Reporting the results of an analysis have a major impact on the value of the tool.

As stated before, static analysis tools have the disadvantage of providing many false positives. This reinforces the need for a human component, therefore increasing the interaction of the user with the tool and reinforcing the need for decent reporting.

Many tools are available as IDE plugins. This is very useful because it can be used the IDE interface to point possible errors directly in the code. Also, developers are already familiarized with the IDE (and possible other plugins), therefore reducing the period of adaptation.

Possible useful reporting features are: grouping, sorting and eliminating results; classify problems by their severity or category; give a description for the problems and provide possible solutions; show the history of analysis made over a web application. As bigger the range of reporting features provided by the tool, the bigger is the probability of the user correctly understanding the problems.

4.3 Open Source Tools

Different tools means different techniques, what usually leads to different results. Particular techniques affect the output of a tool differently and the choice over different techniques it's dictated by the goals of the tool. Depending on the kind of flaws which the tool is intended to find, different techniques may be required. A common technique used by security experts, is to cross-check their reports from multiple tools. There are even some tools like Yasca (described later in this chapter) which automates this process by submitting the code through multiple analysis tools, consolidating the results with well defined metrics and presenting a single report to the user.

Not all the tools described below were designed for security purposes, but they are all useful to developers with security concerns. The study of these tools will be focused on their extensibility to security purposes and in their ability to, within their intended context, be useful in a security analysis.

As stated before, there are a big number of open source static analysis tools, but not all of them are subject of study. In the context of this work, just some of them are described next, being the choice made based on the tool's architecture,

methodologies and techniques. If a tool is here described, it's because it can bring important knowledge and concepts when building the architecture for a new static analysis tool.

4.3.1 FindBugs

FindBugs [*Ayewah et al.* (2008)] is a bung finder tool which performs static analysis over Java bytecode. It works by looking for occurrences of bug patterns in the bytecode. In the context of Open Source tools, FindBugs is the tool most recognized, as it is very flexible and provides the best possibilities to write custom bug detectors.

By its nature, FindBugs is not a security tool. To use it as a security tool, one has to use it out-of-the-box, and write custom security detectors. There are also some security detectors which have been integrated over time and other available online for download. However, build custom security detectors is not trivial and requires a good understanding of FindBugs architecture, therefore requiring experience users.

Because it works over Java bytecode, FindBugs only requires compiled class files to work. However, it allows the user to specify source files to see directly the related suspicious code in the code. Operating over bytecode allows FindBugs, not only to analyze the source code of the application, but also to analyze dependent libraries or any other components previously compiled into bytecode.

FindBugs perform analyses using BCEL[1]. It also extends BCEL in order to provide data flow features. It allows data flow analysis over a control flow graph, and already provides some data flow detectors (for example, for detecting SQL Injection vulnerabilities).

So, FindBugs detectors can be as simple as a visitor pattern over class files, or be more complex and use the control flow graph to perform data flow analysis.

FindBugs has also a very good organization, as each bug pattern belongs to a group (correctness, bad practice, security, etc) and is assigned a priority (high, medium or low). FindBugs is available as a standalone application or integrated into Java IDE's, including Eclipse and NetBeans.

4.3.2 TAJ

TAJ (Taint Analysis for Java)[*Tripp et al.* (2009)] is a tool which detects data flow vulnerabilities using Taint Analysis.

[1] Apache BCEL online page, 2013: "http://commons.apache.org/proper/commons-bcel/"

TAJ is presented here mainly because of its scalability, improving analysis over large applications. By its authors, is described as a tool "designed to be precise enough to produce a low false-positive rate, yet scalable enough to allow the analysis of large applications. TAJ incorporates a number of techniques to produce useful results on extremely large applications, even when constrained to a given time or memory budget. Furthermore, TAJ supports many complex features of Java Platform, Enterprise Edition (Java EE) Web applications".

As any other taint analysis tool, TAJ works based on a set of sources, sanitizers and sinks. It performs the analysis in two stages: first, it performs pointer analysis and builds a call graph. Then, it runs a hybrid thin slicing algorithm to track tainted data. This thin slicing algorithm is the core of TAJ. Thin slicing algorithms are very useful in taint analysis as they allow to capture the statements most relevant to a tainted flow.

That's also the main reason why TAJ is presented here. Thin slicing is a very solid and useful technique that has proven to be capable of improving static analysis efficiency without compromising precision. Before TAJ, a set of context sensitive and context insensitive thin slicing algorithms were used, but TAJ runs a hybrid slicing algorithm in order to get a better trade-off between the number of false positives and the scalability of the algorithm.

4.3.3 Pixy

Pixy [*Jovanovic et al.* (2010), *Jovanovic et al.* (2006)] is another good example of an open source tool capable of detecting XSS vulnerabilities using taint analysis. Although Pixy was developed to find vulnerabilities over PHP applications, the taint-based data flow analysis is the basis for any efficient tool aiming to find vulnerabilities as XSS or Injection.

Pixy is an open source tool for PHP analysis, but it was implemented in Java. It is presented here because of its simplicity and efficiency in applying flow-sensitive and inter-procedural data flow analysis to find security vulnerabilities. Pixy is also the perfect example of the data structures and strategies presented in this chapter. It uses Jflex[1] to build the AST, generates the CFG and then applies a taint based data flow analysis over those structures. However, it does not build the CG, as intra-procedural analysis is no provided.

Pixy is explained by its developers as follows "we use flow-sensitive, inter-procedural and context-sensitive data flow analysis to discover vulnerable points in a program.

[1] Jflex online page, 2013: "http://jflex.de/"

In addition to the taint analysis at the core of our engine, we employ a precise alias analysis targeted at the unique reference semantics commonly found in scripting languages. Moreover, we enhance the quality and quantity of the generated vulnerability reports by employing an iterative two-phase algorithm for fast and precise resolution of file inclusions".

Although Pixy is presented and focused on XSS vulnerabilities, it can be easily extended to find other data flow vulnerabilities like Injection.

4.3.4 F4F

Modern web applications are built using one or more development frameworks. These frameworks simplify the work of a developer, by providing a higher level of abstraction from some implementation details. However, they also represent difficulties in static analysis. Open source tools, normally just ignore the fact that the application is built upon such frameworks, significantly reducing the efficiency of the analysis itself.

In the field of analysis over framework-based web applications, there is F4F (Framework for frameworks) [*Sridharan et al.* (2011)]. There are some previous works on this area but they all are mainly focused on some framework features, with an ad-hoc perspective. F4F provides a more general solution, capable of handling multiple frameworks, which becomes very important with the increasing number of frameworks used today.

By its authors, F4F is described as a "novel solution that augments taint analysis engines with precise framework support and allows for handling new frameworks without modifying the core analysis engine. In F4F, a framework analyzer first generates a specification of an application's framework-related behavior in a simple language called WAFL. The taint analysis then uses the WAFL specification to enhance its analysis of the application".

F4F authors, identify the problem as the framework's use of reflection to invoke application code, based on information provided in the configuration files. This extensive use of reflection causes well known difficulties for static analysis. They also identify that "handling reflection via code analysis can also cause scalability problems, as excessively over-approximate reflection handling can lead to analysis of a large amount of unreachable code."

Besides reflection, F4F authors identify that complex string manipulation and data structures usage in framework code can also cause static analysis difficulties.

F4F works by generating a specification of the program's framework related behaviors, called WAFL (Web Application Framework Language).

This is a great advantage, as new frameworks can be added without changing the core analysis mechanisms. For each framework introduced, only a new WAFL specification needs to be built. Then, that specification is used with the analysis mechanisms and more precise results can be achieved when analyzing framework based web applications.

In F4F, this methodology is used to improve taint analysis and a simple example is built as proof of concept. However, it becomes perfectly clear, that this approach can be easily extended to other static analysis strategies and even other fields of code analysis.

4.3.5 PMD

PMD scans Java Source code, looking for errors that depends on programming syntax. Typically, this errors are sign of bad programming. This kind of errors would be very important when analyzing HTML5 applications. As evidenced before, bad practices in HTML5 can lead to security vulnerabilities and, this syntax analysis, would be very important to avoid those vulnerabilities.

PMD works with one abstract syntax tree built over the source code by a JavaCC (Java Compiler Compiler)[1] generated parser. The AST is based upon an XML schema.

One of the goals of PMD is to make AST walker analysis easy to write. As so, PMD allows the creation of custom rules both in Java or XPath expressions. As it looks only for programming syntax errors, PMD is not very suitable for data flow analysis, significantly reducing the utility of PMD in security analysis.

PMD has a lot of existing rules for many simple and specific situations, thus making it suitable for small and specific analysis. However, it's also very limited in more complex analysis over complete Java EE applications. PMD is available as a command line program or integrated into Java IDE's, including Eclipse and NetBeans.

4.3.6 Yasca

The analysis made by Yasca[2] is different from the analysis made by the set of tools presented so far. However, it's a very useful tool, as it's capable of analyzing

[1] Java Compiler Compiler online page, 2013: "http://javacc.java.net/"
[2] Yasca online page, 2013: "http://www.scovetta.com/yasca.html"

multiple languages (Java, PHP, ASP, C, C++) and it's focused on both bug-finding and security analysis. It's described by the author as a "glorified grep script plus an aggregator of other open-source tools".

Yasca uses external tools to perform the analysis and it's presented here because it is, probably, the most well known and recognized tool with such strategy. Tools like Yasca are very useful because they are able to compile results from multiple tools and show them them to the user in a single report, therefore reducing the effort needed to review the results and, for example, identify false positives.

Yasca started as a set of perl scripts that grepped through source code looking for XSS vulnerabilities. More recently, it became an OWASP tool[3].

Its main characteristic is to be highly extensible, as it is built in a plugin-based architecture. Originally, it contains a set of custom scanners built just for Yasca.

To perform analysis, the user can build it's own rules or even integrate external tools. The external tools mechanism, uses a set of external tools (for example, FindBugs, PMD, JLint, Javascript Lint, PHPLint, Cppcheck, ClamAV, RATS or Pixy) to perform the analysis.

4.3.7 PQL

PQL (Program Query Language) [*Martin et al.* (2005)] is another solution very different from the set of tools presented so far.

PQL is not a tool, is a query language which allows users to express a large class of application security patterns. Then, PQL system, based on those security pattern, automatically generates a pair of static and dynamic checkers.

The static checker has the goal of finding all the potential matches of the pattern in the code.

The dynamic checker has the goal of identifying errors precisely when they are found (by the static checker) and enabling actions like logging, recovery or other user specified actions.

In Figure 4.6, a simple PQL query from [*Martin et al.* (2005)] is shown. By analyzing the query, it's possible to easily understand that the query is named "forceclose", looks for situations where InputStream objects are instantiated but not closed before the end of the method (being this the static part). Then, if a situation occurs, a call to close() is inserted automatically (being this is the dynamic part).

In [*Martin et al.* (2005)], PQL is implemented in a proof of concept tool and used to perform analysis over six real world web applications. Also, there are some

[3]OWASP Yasca project, 2013: "https://www.owasp.org/index.php/CategoryOWASP_Yasca_Project"

literature and small solutions which explore PQL in the development of static analysis tools.

```
query forceClose()
uses object InputStream in;
within _ . _ ();
matches {
    in = new InputStream();
    ~in.close();
}
executes in.close();
```

Figure 4.6: Example of a PQL query.

4.4 Summary

This chapter completes another goal proposed for this dissertation: perform a detailed analysis over static analysis tools, methodologies and strategies.

During this chapter, it was studied the main stages of the process of the developing a static analysis tool targeting security. For each stage, it were described the most important data structures needed and the strategies that must be used.

Then, this was consolidated with the analysis of a set of frameworks and open source tools. The goal was to understand the work that already has been done in this field, in order to better understand what can be accomplished in the future.

In the next chapter, based on the knowledge acquired from chapters 2 and 3, and the understanding of static analysis tools acquired from this chapter, it will be defined a set of guidelines and a high level platform for a static analysis tool.

CHAPTER V

An Approach to Handle Static Analysis

This chapter is where all the information presented so far is compiled together. Basically, it will be answered questions like: What's missing on open source static analysis tools? Do the existing tools cover the main threats on web security today? What can be added to the existing tools and methodologies? Are the today tools and methodologies useful when dealing with HTML5 security? This chapter has the main goal of putting the right questions and simultaneously providing possible implementations and solutions. Then, those solutions will be compiled into the form of a high-level approach for a platform capable of performing static analysis.

5.1 Basics

Modern web applications are multi-tier systems built with multiple languages and technologies. Taking as an example, a simple JEE application is normally composed with a data layer (normally SQL databases), a business Layer (the JEE part) and a presentation layer (HTML, Javascript, etc). There are also the integration of external libraries, communication with external systems (web services, for example), and many other functionalities.

So, and taking into account all that has been said in the previous chapters, for an application to be capable of parsing and analyzing the security of a web application, it has to be built with consideration of the vulnerabilities that it targets. Vulnerabilities are different and need different approaches to be found.

To find SQL injection vulnerabilities, for example, it's necessary to follow and

analyze the application data flow, in order to understand if malicious instructions are able to get to the database and be executed as commands. On the other hand, to detect security misconfiguration errors and bad practices, data flow analysis is not needed and a simple pattern-based analysis is enough.

In chapter 3, HTML5 security was reviewed and possibly vulnerabilities were discussed. The main conclusion from chapter 3 is that HTML5 will introduce two different threat vectors. It will boost XSS vulnerabilities, as more information is being used on client side, and it will introduce new possible attack vectors that can lead to new vulnerabilities and to new ways of exploring old vulnerabilities.

So, reviewing the security of a web application built with HTML5, is a two step process. First, it has to be ensured that the application has no XSS vulnerabilities. If one application has XSS vulnerabilities, then almost every feature of HTML5 is compromised. The second step is to ensure that the most critical features are properly implemented. As stated before, HTML5 was developed with a great concern on security (probably like any language before). Each feature has a chapter of security considerations in the specification. If followed, those considerations will lead to secure implementation of the features. On the other hand, poor implementation will almost definitely lead to very serious problems.

The first step depends mainly a taint based data flow analysis, finding possible sinks where input can get without being properly validate. In chapter, multiple tools multiple capable of finding XSS vulnerabilities were presented and most of them used taint based approaches.

The second step requires a pattern based analysis, looking for possible poor implementations. Taking the most trivial and dangerous example, the CORS header defined like this:

```
Access-Control-Allow-Origin: "*"
```

Listing 10: Bad implementation of a CORS header

Finding this kind of errors is as simple as looking for in HTML pages for implementations like this one. The same happens with the poor integration of new tags and attributes, and many other errors. The solution is to compile all security considerations surrounding HTML5, transforming them into patterns and look for those patterns in HTML pages.

5.2 Features

After the analysis made over the open source static analysis tools, there were some limitations that can be highlighted as the most common between all the tools.

Next, this limitations are presented as a list of features, that are not totally integrated in the open source tools but which are very important to consider when performing analysis over modern and complex web applications.

5.2.1 Security Patterns

Software design patterns are reusable solutions to commonly occurring problems in the design of software applications. They take advantage of the experience and knowledge acquired during the development of a specific solution. A design pattern describes the solution, the problem and the problem in which it must be used.

The same concept of software design patterns, can also applied to the security of web applications, as security design patterns. This is not new, there are multiple academic works in this field, for example [*Dalai and Jena* (2011)], where design patterns are defined and put into use in terms of security.

Security design patterns are also the key to find HTML5 poor implementation vulnerabilities. The tool to build must be able to receive a set of security design patterns and find them in the code. An important work has also to be made in defining the security patterns. HTML5 security considerations is a good place to start. Each consideration is a candidate to be built as a security pattern. In the perfect scenario, all security considerations identified in the specification, would have the corresponding security pattern. But, of course this is not a reality as there are security considerations to abstract and others to complex to integrate into patterns. This is an approach very similar to PMD (tool presented in chapter 4). As it will be shown below, PQL (also presented in chapter 4) can also be very useful in this kind of analysis.

5.2.2 Context Sensitive

Generic code usually means generic flaws, common to multiple applications. These kind of flaws can be found by applying the same analysis methods on all the applications. With the growing use of frameworks and design patterns, applications are becoming more and more generic, and the amount of custom code per application is significantly reducing.

However, there are still a big number of applications built from scratch using custom code. And, of course, there are still some features which need custom implementations. In these situations, it may occur some flaws which are specific to an application. These flaws are considered context-specific and usually require customization over the analysis methods.

In [*Chess and West* (2007)], this situation is demonstrated with an example of a program that handles credit card numbers. To comply with the payment card industry data protection standard, a credit card number should never be completely displayed back to the user. There is no standard implementation to this feature, and so, different applications will have different implementations of it. Therefore, find specific flaws in this situation, would require understanding specific functions and data structures used in the application code.

Concluding, the approach to considerer when reviewing an application's source code, depends on whether they are generic or context specific. This is one of the biggest challenges when building a static analysis tool. There are some scientific approaches to build more context-sensitive solutions, but the best solution to approach specific situations is to allow the user to customize his own rules. In fact, in the open source world, this solution is consensual between almost every tool.

This extends the approach previously made surrounding security patterns. If the user is able to build their own patterns, and integrate them in the tool, the application can grow with the users. A very good example is FindBugs. Many detectors available for FindBugs today, are design by users and then suggested to the development team. This allow the tool to grow with experience and to evolve as the methodologies evolve. As so, not only the tool must let the user to build their own patterns, as also must place the proper mechanisms for users to suggest new patterns into the core of the tool.

There should also be noted that custom patterns does not compensate for tool limitations. Custom patterns are restrained by the limitations of the tool. If a tool doesn't perform data flow analysis, then there no way of building custom patterns capable of efficiently detecting vulnerabilities like XSS or Injection.

5.2.3 Modularity

There are more than 100 open-source tools and they all have different characteristics and goals. Each one of them covers a set of vulnerabilities, but none of them covers them all, as that would be almost impossible to achieve. Mainly because different kind of vulnerabilities have different characteristics and need different strategies

to be found.

Earlier in this document, it was already explained that are tools which try to fill this gap, by aggregating a set of another tools, taking advantage of their different strategies and methodologies. A good example is YASCA, presented in chapter 4.

Static analysis tools can then be divided into two categories: a tool which is built specifically to a set of vulnerabilities, very efficient withins its goals, but unable to be adapted and expanded. Or a tool capable of analyzing a wider set of vulnerabilities but that is not so efficient in each one.

To achieve modularity, each component of the solution must be built to a specific purpose, but must also be built in order to be extended. This thinking must be putted in every component and in every consideration. Besides the fact that this work is mainly focused on Java and HTML5, multi language analysis, communication services and frameworks will be approached next. The goal is to design a modular tool capable of growing as a whole and integrate new modules over time.

5.2.4 Multi Language Analysis

Multi language analysis can be seen in two different ways. First, the ability of the tool to parse and analyze applications with different languages (for example, a Java application and a .NET application). On the other hand, most of the modern web applications are composed of multiple modules built upon various programming and specification languages. As stated before, for example, even the most simple Java web application may consist of SQL, Java, HTML and Javascript. This can also be seen as multi language analysis, but, for simplicity purposes, it will be referred as mixed language analysis.

In general, open source analysis tools cannot process these multiple or mixed languages systems as a whole, because they are built to a particular language. At best, they can process multiple languages, but do it individually, or by integrating external tools.

The only way to perform analysis over applications which integrate multiple languages or consist of multiple heterogeneous processes, is to build common models. Getting out of a the static analysis context, there are approaches specially designed to abstract particular aspects of languages into common models. A good example is [*Strein et al.* (2006)], where this problem is identified as "the lack of a common meta-model capturing program information for analysis and refactoring that is common for a set of programming languages abstracting from details of each individual language, and that is related to the source code level of abstraction in order to allow

for source-code analysis and refactoring".

To approach this problem, in [*Strein et al.* (2006)], a common meta-model architecture is designed and applied into an IDE, built as proof of concept. The common meta-model captures the information, and represent it in a language independent structure. To build this representation, first, multiple specific front-ends build language specific meta-models. Then, parts that are relevant are abstracted to a common model.

In Figure 5.1, is presented the kernel architecture proposed in [*Strein et al.* (2006)]. The same concepts seen in Figure 5.1, can be applied to any application which an-

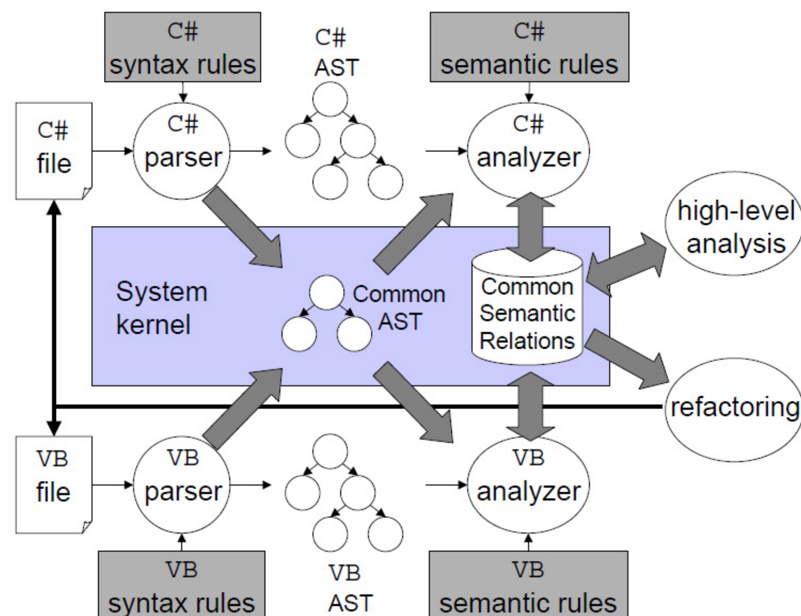


Figure 5.1: Kernel architecture, [*Strein et al.* (2006)].

alyzes code. The disadvantage of building common models is the lost of precision. Sometimes is impossible to abstract some specificities into those models, or that will lead to the lost of important information for the analysis.

In the world of proprietary tools, there are multiple applications who process systems with multiple or mixed languages. However, in open source tools this is not available yet. There are some tools starting to draft some concepts over mixed language systems, but there isn't a single open source tool who performs this as proprietary tools do.

Drilling down on the architecture from Figure 5.1, the question is how can a static analysis tool leverage from a system like this to perform multi-language static analysis. There are multiple academic approaches to this, and many prototypes built

as proof of concept, but the fact is that none of them is yet included in open source tools.

In these systems, it becomes clear that there two type of IR models: language-specific or common models. It's hard to perform an efficient pattern based analysis for Java vulnerabilities over the common models. This kind of analysis must to be done over the specific models. But, an efficient tainted-based data flow analysis can be applied just by identifying the source, sink and derivation descriptors to track the information flow and so, it can be made over the common models ans still present reliable results.

A language is no more that a grammar applied into work. As so, by having access to the language grammar, as explained in chapter 4, it's possible to parse the code into a series of models. The process is the same for different languages. These models can then be abstracted to common models, and the analysis made over models. In the case of an inter and intra procedural-data flow analysis, these models are the AST, the CFG and the CG.

Of course this is only possible in similar languages like Java and .NET. It's not possible to build common models between HTML5 ad Java, because the languages have to be semantically alike.

This answer the question of how to build an application not restricted to a particular language. But, systems like the one presented in Figure 5.1, have the limitation of being unable to perform analysis in systems with mixed-language analysis.

Taking as example Javascript, which has very similarities with Java. The exact same process used to parse a Java program, and build the IR models, can be used to build the models in Javascript. Then, the same type of analysis can be performed over those models. However, this is still a case of multi-language analysis. The problem is to analyze mixed-languages (in this case, Java and Javascript) in the same application and the interaction between them. This is a completely new field, very complex (again, with multiple scientific approaches being done) but that will be left out of the context of this work.

5.2.5 Frameworks

During the course of this document, it was already highlighted the importance and popularity which development frameworks have in today's software development process.

Even though they are so important, the fact is open source tools for security analysis, completely ignore development frameworks. Some of them, allow the user

to write their own rules, which gives the user some liberty to built rules which are capable of understanding development frameworks. On the other hand, this is not always true, because many of the tools provide their own rules mechanism, with their own language. This mechanisms, can limit the user in a way that he can not build rules capable of understanding anything outside a set of specified functionalities.

Frameworks are much like languages, they have a specification, which the developer must use and understand in order to perform actions. As tools need to understand the language's specification to be able to analyze it, the same thing happens with development frameworks. Tools need to understand the framework's specification, in order to make analysis over it.

Of course this is not as simple as it sounds. Frameworks introduce a all new level of complexity, mainly because they are becoming more and more user-friendly. Features like configuration files, annotations, abstractions, etc. All the choices presented so far in this chapter will lead to the execution process presented , make programing easier to the developer, but harder for a tool to analyze it. However, if configured properly, as it becomes easier to the developer, it also gives him less responsibility, reducing significantly the risk of him introducing vulnerabilities into the application.

A good example is SQL injection. When building the ORM (object/relational mapping), the proper use of prepared statements it's essential in order to avoid SQL injection. However, a less experienced developer, or even a simple mistake can lead to a SQL injection vulnerability. ORM frameworks, if used and configured properly, can completely remove this risk. A very good example is the Hibernate[1] framework, which uses transactions and prepared statements in every one of the operation it does, therefore removing the chance for a SQL injection vulnerability.

This opens a wide range of possibilities on how should development frameworks be approached, when building a security analysis tool.

A possible solution is to analyze the framework code, annotations and configuration files in order to find situations of improper use of the framework. This is a good approach, but it needs a great level of framework knowledge, and a mechanism capable of comprehend and analyze plain text, configuration files and language code, all together. This would also make the tool much framework-specific, making it hard to extend the solution to multiple frameworks and forcing the limitation to a small set of frameworks.

Another possibility, much like F4F (presented in chapter 4), is to remove the framework specific code from the picture. This approach has the advantage of mak-

[1]Hibernate online page, 2013: "http://www.hibernate.org/"

ing it easy to extend the solution to multiple frameworks. However, it puts a lot of responsibility on the compiler mechanism. F4F solves this by building a framework analyzer which generates a specification of the application's framework-related behavior in a simple language called WAFL. The taint analysis then uses the WAFL specification to enhance its analysis of the application. The great advantage of this approach is that, based on the frameworks's specification file (WAFL), it's possible to remove the dependency from framework-specific code, and integrate it directly into the application's source code.

5.2.6 Web services

In modern applications, web services are very common as a way to communicate with applications inside or outside the application network. If the service code is provided, the same approach taken to parse frameworks, can be applied into web services (or other similar communication services, including HTML5 web sockets). The WSDL (the web service descriptor) and other configuration files (that probably will be part of a framework used) can be parsed in order to understand the service to be invoked. Then, the path to the service implementation can be calculated and. If the source code is available, the service call can be treated as a regular function and included in the CG. In this case, relation and invocations of the web service can be threated as normal function invocations and data flow analysis can be made.

When external web services are invoked, and there is no access to the source code, they must be considered as output (when the web service is invoked) and input (when the application receives data from the web service).

5.2.7 Reporting

Previously in this document, it was already highlighted the importance of a quality reporting. It's not enough for a tool to efficiently find security vulnerabilities, if it fails to pass that message to the user. As so, reporting is a very important part of any tool, and it's clearly overlooked by security open source tools.

It was explained before in this document that one of the disadvantages about static analysis tools is the lack of precision, because they present many false positives. Also, there are always some reported situations which users, for different reasons, may not want to consider fixing.

This reinforces the importance for the user to be able to parametrize his reporting, specially in big applications. As the complexity of applications grows in size, so do

the number of possible vulnerabilities reported, making even more essential for the user to be able to filter the information.

The best approach is for a vulnerability to be classified in two strands: category and severity. This will allow the users to select what vulnerabilities they want to catch or ignore or highlight, based on their category or severity.

This is not new at all. There are already several tools and proposals which allows the user to browse bugs/vulnerabilities by their category, severity or both. These is a very important feature for developers to define their strategies and methodologies in their search for flaws.

A feature not usualy seen in open source tools is application history. By being able to compare current reporting with previous versions, users can many times find patterns in the their code, which can help them correcting methodologies, identify false positives or even understand that the one they were considering as a false positive, afterwards is not.

5.2.8 Software as a Service

As stated before, business companies moved their entire systems to the web and people are moving their day-to-day operations to the web. This is all part of a paradigm shift called cloud computing [*Zissis and Lekkas* (2012); *Qian et al.* (2009); *IBM* (2009)].

The best approach is for the solution to be available as cloud service. The application will run entirely in the cloud and it will be delivered to the client as a service ready to use, in a browser, removing any installation responsibility or maintenance from the client. In practice the user will access the platform, submit his code for review sand receive a detailed report about the vulnerabilities in the code.

5.3 Execution Process

Based on the methodologies and tools presented in chapter 4, and all the choices and features presented above in this chapter, it will be now described the general architecture and processes of the solution proposed. Figure 5.2 presents a very high-level scheme of the global execution process. Next, each of the stages from the schema will be detailed.

5.3.1 Execution flow

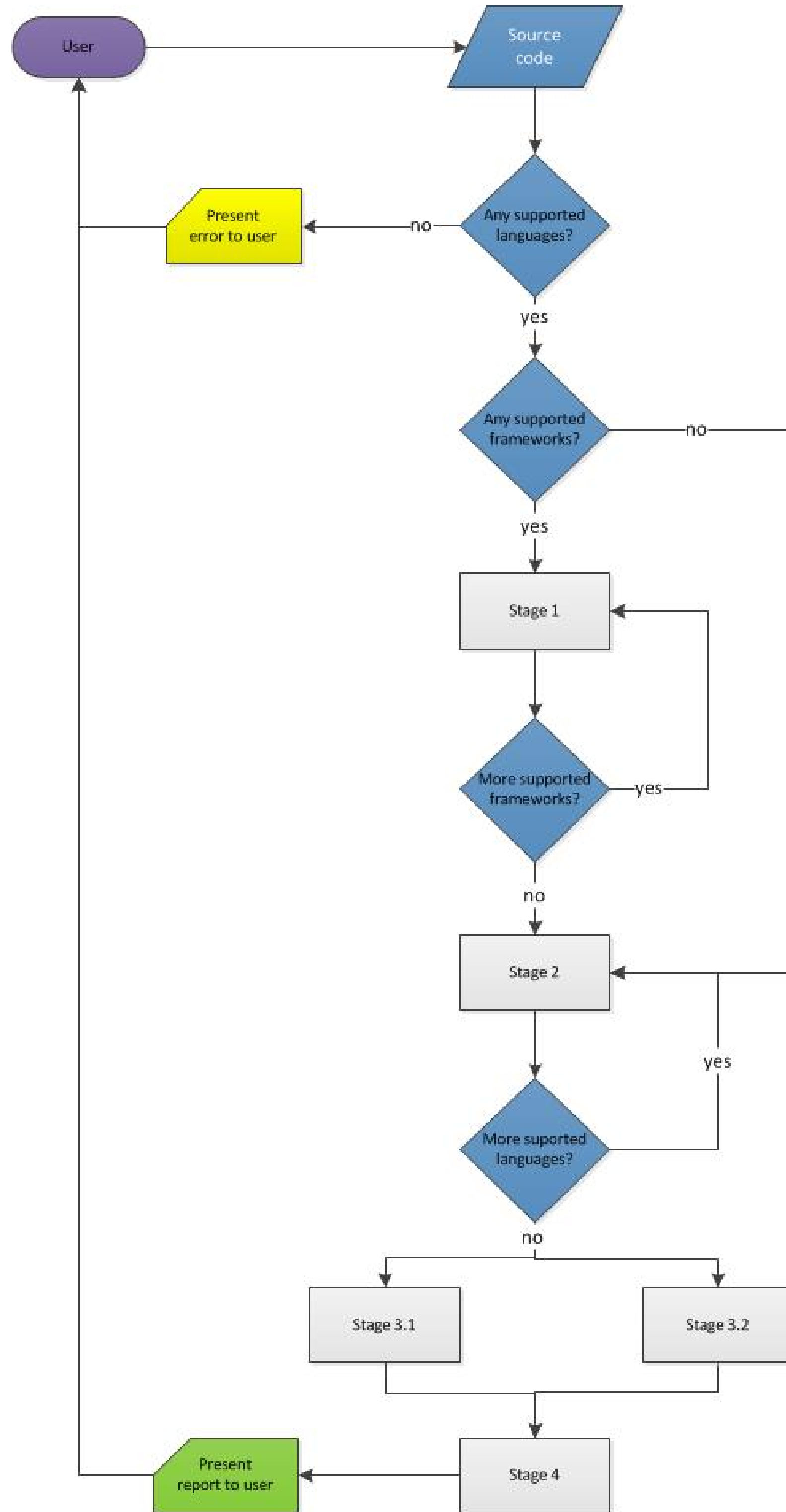


Figure 5.2: Global Workflow.

5.3.2 Stage 1

User Input:	Source code
System input:	Framework descriptors
Output:	Source code independent from the framework

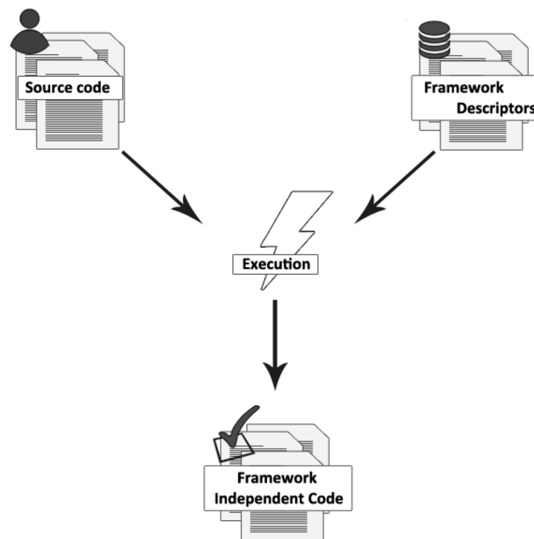


Figure 5.3: Diagram of stage 1.

In this stage, the goal is to make the source code independent from the framework, in which the application was built.
If a framework is not recognized by the system (the descriptor for that framework doesn't exist), the framework must be ignored and the analysis must continue to the next stage.

The application in analysis may have been built using multiple frameworks. In these cases, this stage must be applied one time for each framework, sequentally.

For example, let's suppose that one application is built with Spring[1] framework for Java, and jWebSocket[2] for web sockets. In this case, Spring must be processed first and only then jWebSocket. The simplest way to do that is to give weights to each category of frameworks.

If, for some reason, the code processed is not in accordance to the framework descriptor, the user must be notified, the framework ignored, and the analysis must continue to the next stage.

[1]Spring online page, 2013: "http://www.springsource.org/spring-framework"
[2]jWebSocket online page, 2013: "http://jwebsocket.org"/

This will also ensure that, by processing one framework first, the correctness of a second framework is compromised, then, the second framework will be ignored, the user notified, and the analysis must continue to next stage.

5.3.3 Stage 2

User Input:	None
System input:	Language descriptors
	Framework independent code (output from stage 1)
Output:	IR models

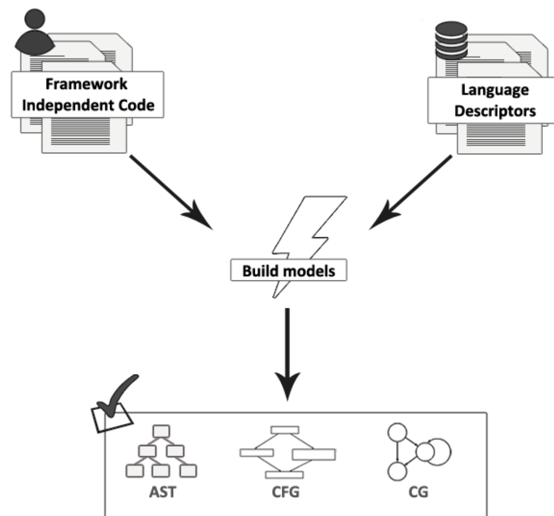


Figure 5.4: Diagram of stage 2.

In this stage, the goal is to build all the models needed for analysis. In chapter 4, each one of these models and data structures are explained in detail.

If the program is built over multiple languages, this stage must be done sequentially. Like in the previous stage, the order in which the languages are processed should be determined by its importance.

For example, if an application built in Java with HTML and Javascript, Java must be processed first. As stated before, this is not ideal, as the interactions and relationships between those languages should be target of analysis. However, this mixed-language systems fall out of the context of this work.

If the application was built with multiple languages, and one of them is not recognized by the system, then that language must be ignored and the other ones processed normally. If none of the languages are recognized, the analysis must stop and not proceed to the next stage. Also, the user must be notified that the system is not capable to perform the analysis on the application.

One the main goals of the solution is for it to be able to analyze Java and HTML5 at the same time. With this approach this is possible as Java will be parsed and the corresponding IR models built. Then, sequentially, HTML5 will be independently parsed and the corresponding IR models built.

Parsing the application source code is a process performed by multiple tools and frameworks. Some of them are open source and very easy to use and extend. Are example of such tools: ANTLR[1], JavaCC[2] (used by PMD), BCEL[3] (used by FindBugs) and many others.

5.3.4 Stage 3.1

User Input:	None
System input:	Security Patterns
	IR models (output from stage 2)
Output:	Analysis results persisted on DB

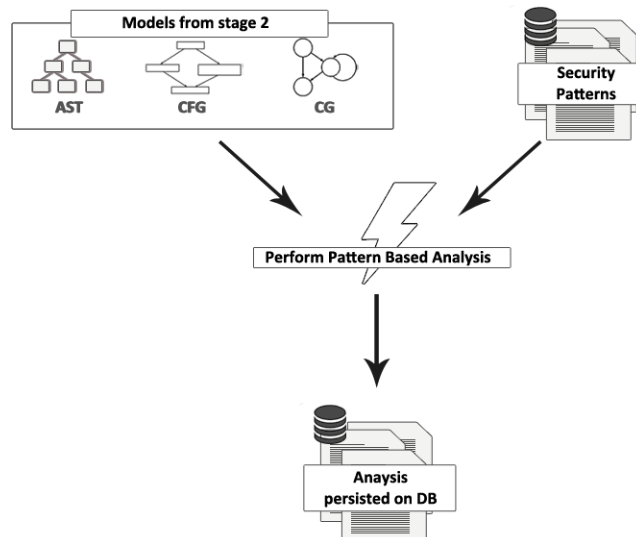


Figure 5.5: Diagram of stage 3.1.

[1]ANTLR - ANother Tool for Language Recognition online page, 2013: "http://www.antlr.org/")
[2]JavaCC - Java Compiler Compiler online page, 2013: "http://javacc.java.net/"
[3]Apache BCEL online page, 2013: "http://commons.apache.org/proper/commons-bcel/"

After building the IR models, the execution process takes two different paths, corresponding with the two step process described in the basics of this chapter.

This is the stage where a pattern based analysis is made, in order to find bad practices and coding errors that may lead to security vulnerabilities. The goal is to verify the security patterns against the IR models generated in the previous stage. The quality of the analysis will be proportional with the quality the security patterns.

This stage can be simplified by using PQL, the query language presented in chapter 4. PQL system performs the analysis based on PQL queries. In this case, the security patterns would be built in the form of PQL queries. So, by using PQL system, there is not much to be done besides building the security queries or let the user build them.

Of course there is also an integration effort in order to integrate PQL into the application core. This integration consists mainly on adapting the IR models from the previous stage, in order to pass them as input to PQL system.

Of course, using PQL is only one option. There are multiple pattern-based analysis algorithms, and implementing from scratch is always a possibility.

In the end of this stage, the analysis results will not be directly generated into a report. Instead, the results will be persisted into the database (where can also be the results of previous analysis on the same application). These records will be processed in stage 4.

5.3.5 Stage 3.2

User Input:	None
System input:	Language Descriptors
	IR models (output from stage 2)
Output:	Analysis results persisted on DB

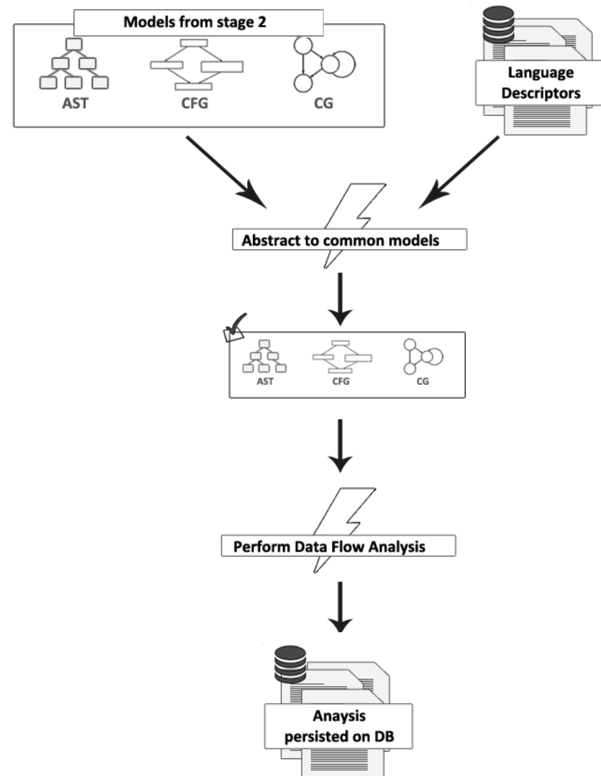


Figure 5.6: Diagram of stage 3.2.

This stage performs taint based data flow analysis and has the main goal of finding XSS vulnerabilities. Like any other data flow analysis, this stage can be easily extended to identify other data flow vulnerabilities.

As stated before, in order to this stage to be easily extended to multiple languages, the IR models must be first abstracted into common models. In practice, this means that all specifics from the language must be removed. With this approach, the same analysis methodologies can be applied over Java or .NET, for example.

To detect XSS vulnerabilities, an intra and inter procedural taint based analysis is then applied over the common models. More advanced algorithms (like the thin slicing used in TAJ), can also be considered to improve efficiency.

As in stage 3.1, the analysis results will be persisted into the database and not directly compiled into a report. These records will be processed in stage 4.

User Input:	None
System input:	Previous analysis
	Current Analysis (output from stage 3)
Output:	User report

5.3.6 Stage 4

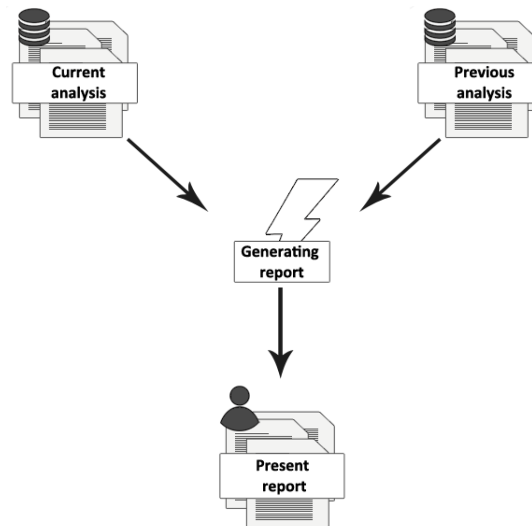


Figure 5.7: Diagram of stage 4.

The goal in this stage is to build the final report which will be presented to the user. To do this, the current analysis must be compared to the last analysis performed over the same application. By doing this, it will be possible to show the user the evolution achieved over time. For example, the user will be able to see what vulnerabilities he has fixed, what vulnerabilities are new and what vulnerabilities he forgot or ignored.

Also, like stated before, vulnerabilities must be characterized by their severity and category, allowing the user to easily filter the information.

Decent open source reporting tools are available. BIRT[1] and JASPER reports[2] are two very good examples, and can be very useful in this stage. They include very important features, like user basic interaction, graphics, charts, multi-format exporting, web based reporting, etc.

[1] BIRT online page, 2013: "http://community.jaspersoft.com/project/jasperreports-library"
[2] Hibernate online page, 2013: "http://www.eclipse.org/birt/phoenix/"

5.4 Summary

During this chapter, a set of guidelines were defined over some features considered critical when building a static analysis tools. These features are usually not seen in the open source static analysis tools available today and are essential to build a platform capable of performing analysis over a modern and complex web application:

- Be developed as a software as a service.
- Be interactive and integrate good reporting services.
- Be context-sensitive.
- Be built upon security patterns.
- Be capable of analyzing applications built with development frameworks.
- Be capable of analyzing applications built upon mixed-languages.
- Support multiple languages.

The list of features presented are essential for a tool to keep up with the constant changing in web applications and web development.

Was also presented and described a high-level proposal for the execution process of a platform capable of analyzing the security of a web application. It's a process with four main stages, each one described more detailed, and capable of integrating multiple languages, multiple development frameworks and to perform two types of analysis: data flow and pattern based analysis.

Another conclusion from this chapter and from this work is that reviewing the security of a HTML5 application is a two step process. First, a tainted data flow analysis must be done over the business layer in order to identify XSS vulnerabilities. The second step is to perform a pattern based analysis over the presentation layer to ensure that the most critical features of HTML5 are properly implemented, and follow the security considerations presented in the specification.

CHAPTER VI

Conclusions

This work starts by making a detailed survey about web security, the main vulnerabilities seen on web applications today, and how they are evolving with the emerging of new technologies, specially the new HTML5 specification. Then, a complete analysis over static analysis tools and techniques is made, in order to understand the process of building a static analysis tool. It became clear what is missing on open source tools, and how can the already existent methodologies be used in HTML5 security analysis.

One biggest conclusion which comes out of this work is that one application containing HTML5 is only as secure as the business layer is, and the main threat against HTML5 does not come from the HTML5 specification itself but from XSS vulnerabilities. So, one application looking only for HTML5 specific vulnerabilities may lead to a false sensation of security. Reviewing the security of HTML5 applications is a two step process:

The first step is to ensure that no XSS vulnerabilities are present in the application. XSS vulnerabilities are considered data flow vulnerabilities and so, to find XSS, data flow analysis must be done. To perform data flow analysis, a set of three main IR models must be built (at least), and algorithms like pointer analysis, single statement assessment or taint analysis must be used based on top of those IR models.

The second step, is to find HTML5 bad practices and coding errors. As HTML5 is, by nature, a secure language the main source of security vulnerabilities introduced by this new language will be programmer coding errors. To find this coding errors, a pattern based analysis must be done, looking for patterns in the code that are known to introduce security vulnerabilities.

Modern web applications are complex systems, built with different languages and

technologies. This reinforces the need for static analysis not to ignore the multi and mixed languages systems, the frameworks in which the application was built, communication services (like web services) or even the integration of external libraries and features. To contemplate these features, more advanced static analysis techniques are also needed.

6.1 Output

At the beginning of this work, two main objectives were defined: the first one was to provide a survey on the state of web security, understand the security vulnerabilities currently seen in web applications and understand how they are evolving with the emerging of new technologies and methodologies, specially HTML5. The second objective was to define a set of guidelines and a proposal for a high level model of a platform capable of perform static analysis over complex and modern web applications.

The first contribution is divided between chapters 2 and 3. Chapter 2 starts by analyzing the OWASP Top 10 and all the main vulnerabilities seen on web applications today. Then, in chapter 3, a detailed analysis over the new features of HTML5 is made along with a security analysis over the possible new attacking vectors introduced by HTML5.

The second contribution is mainly made in chapter 5. However, chapter 4 is very important because it establishes the background for all the strategies, methodologies and tools mentioned in chapter 5. During chapter 5, a set of features were identified as being the most important to allow a static analysis tool to keep up with the growing complexity of web applications. Finally, it was presented a very high-level architecture, divided in four stages, and capable of integrating multiple languages and multiple development frameworks. The process is divided mainly between two methodologies: it uses data flow analysis mainly over the business layer of the application (which will allow it, for example, to identify vulnerabilities like Injection or XSS) and uses pattern based analysis mainly over the presentation layer (which will allow it, for example, to identify vulnerabilities due to HTML5 poor implementation).

6.2 Future Work

In the solution proposed in this document, only static analysis specifics (like data structures, algorithms and methodologies) are specified, but implementation details

(like the architecture of the application, database schemas, etc.) were left out.

The next step would be do put this approach into the form of an application, defining all the implementation details and solving all the possible problems that could come up within that process.

Also, during the course of the document, two areas were identify as being very important but were left out, due to their complexity. The first is mixed-systems analysis, which is needed in almost every modern web applications. The second is reverse engineering techniques for program refactoring, in order to be able to recover code that is either not available or compiled. This is specially important when external libraries are used in a web application. With reverse engineering techniques, these libraries could be decompiled into source code, and the analysis could be extended to the library itself.

These two areas are very important, but also very complex, and each one would require it's own independent research. As so, for future work, it would be interesting to study a possible integration of these two concepts into the solution presented.

Bibliography

Aho, A. V., R. Sethi, and J. D. Ullman (1986), *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Artho, C., and A. Biere (2005), Combined static and dynamic analysis, *Electron. Notes Theor. Comput. Sci.*, *131*, 3–14, doi: http://dx.doi.org/10.1016/j.entcs.2005.01.018.

Ayewah, N., D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh (2008), Experiences using static analysis to find bugs, *IEEE Software*, *25*, 22–29, special issue on software development tools, September/October (25:5).

Cannings, R., H. Dwivedi, and Z. Lackey (2008), *Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions*, 1st ed., McGraw-Hill, Inc., New York, NY, USA.

Chess, B., and J. West (2007), *Secure programming with static analysis*, first ed., Addison-Wesley Professional.

Dagenais, B., and L. Hendren (2008), Enabling static analysis for partial java programs, *SIGPLAN Not.*, *43*(10), 313–328, doi:10.1145/1449955.1449790.

Dahbur, K., B. Mohammad, and A. B. Tarakji (2011), A survey of risks, threats and vulnerabilities in cloud computing, in *Proceedings of the 2011 International Conference on Intelligent Semantic Web-Services and Applications*, ISWSA '11, pp. 12:1–12:6, ACM, New York, NY, USA, doi: http://doi.acm.org/10.1145/1980822.1980834.

Dalai, A. K., and S. K. Jena (2011), Evaluation of web application security risks and secure design patterns, in *Proceedings of the 2011 International Conference on Communication, Computing & Security*, ICCCS '11, pp. 565–568, ACM, New York, NY, USA, doi:10.1145/1947940.1948057.

Garcia-Alfaro, J., and G. Navarro-Arribas (2007), Prevention of cross-site scripting attacks on current web applications, in *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part II*, OTM'07, pp. 1770–1784, Springer-Verlag, Berlin, Heidelberg.

Ge, X., K. Taneja, T. Xie, and N. Tillmann (2011), Dyta: dynamic symbolic execution guided with static verification results, in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 992–994, ACM, New York, NY, USA, doi:http://doi.acm.org/10.1145/1985793.1985971.

Haldar, V., D. Chandra, and M. Franz (2005), Dynamic taint propagation for java, in *Proceedings of the 21st Annual Computer Security Applications Conference*, pp. 303–311, IEEE Computer Society, Washington, DC, USA, doi: 10.1109/CSAC.2005.21.

IBM (2009), Ibm point of view: Security and cloud computing, *Tech. rep.*, IBM.

Jim, T., N. Swamy, and M. Hicks (2007), Defeating script injection attacks with browser-enforced embedded policies, in *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pp. 601–610, ACM, New York, NY, USA, doi:http://doi.acm.org/10.1145/1242572.1242654.

Jovanovic, N., C. Kruegel, and E. Kirda (2006), Precise alias analysis for static detection of web application vulnerabilities, in *Proceedings of the 2006 workshop on Programming languages and analysis for security*, PLAS '06, pp. 27–36, ACM, New York, NY, USA, doi:10.1145/1134744.1134751.

Jovanovic, N., C. Kruegel, and E. Kirda (2010), Static analysis for detecting taint-style vulnerabilities in web applications, *J. Comput. Secur.*, *18*, 861–907.

Khedker, U., A. Sanyal, and B. Karkare (2009), *Data Flow Analysis: Theory and Practice*, 1st ed., CRC Press, Inc., Boca Raton, FL, USA.

Kuppan, L. (2010), Attacking with html5, *Tech. rep.*, Attack & Defence labs.

Lee, C. A. (2010), A perspective on scientific cloud computing, in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pp. 451–459, ACM, New York, NY, USA, doi: http://doi.acm.org/10.1145/1851476.1851542.

Lopes, R., D. Vicente, and N. Silva (2009), Static Analysis Tools, a Practical Approach for Safety-Critical Software Verification, in *ESA Special Publication*, *ESA Special Publication*, vol. 669.

Louw, M. T., and V. N. Venkatakrishnan (2009), Blueprint: Robust prevention of cross-site scripting attacks for existing browsers, in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pp. 331–346, IEEE Computer Society, Washington, DC, USA, doi:10.1109/SP.2009.33.

Martin, M., B. Livshits, and M. S. Lam (2005), Finding application errors and security flaws using pql: a program query language, *SIGPLAN Not.*, *40*(10), 365–383, doi: 10.1145/1103845.1094840.

McArdle, R. (2011a), Html5 overview:a look at html5 attack scenarios, *Tech. rep.*, ENISA - European Union Agency for Network and Information Security.

McArdle, R. (2011b), Html5 overview: A look at html5 attack scenarios, *Tech. rep.*, Trend Micro.

Michael Schmidt, C. S. A. (2011), Html5 websecurity, *Tech. rep.*, OWASP - Open Web Application Security Project.

Mui, R., and P. Frankl (2011), Preventing web application injections with complementary character coding, in *Proceedings of the 16th European conference on Research in computer security*, ESORICS'11, pp. 80–99, Springer-Verlag, Berlin, Heidelberg.

Nilsson-Nyman, E., G. Hedin, E. Magnusson, and T. Ekman (2009), Declarative intraprocedural flow analysis of java source code, *Electronic Notes in Theoretical Computer Science*, *238*(5), 155 – 171, doi: http://dx.doi.org/10.1016/j.entcs.2009.09.046, ¡ce:title¿Proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008)¡/ce:title¿.

OWASP (2010), Owasp top 10 - 2010: The ten most critical web application security risks, *Tech. rep.*, OWASP - Open Web Application Security Project.

OWASP (2011), Next generation web attacks – html 5, dom(l3) and xhr(l2), *Tech. rep.*, OWASP - Open Web Application Security Project.

OWASP (2013a), Authentication cheat sheet, *Tech. rep.*, OWASP - Open Web Application Security Project.

OWASP (2013b), Html5 cheat sheet, *Tech. rep.*, OWASP - Open Web Application Security Project.

OWASP (2013c), Session management cheat sheet, *Tech. rep.*, OWASP - Open Web Application Security Project.

Parr, T. (2007), *The Definitive ANTLR Reference: Building Domain-Specific Languages*, Pragmatic Programmers, first ed., Pragmatic Bookshelf.

Pistoia, M., S. Chandra, S. J. Fink, and E. Yahav (2007), A survey of static analysis methods for identifying security vulnerabilities in software systems, *IBM Syst. J.*, *46*(2), 265–288, doi:10.1147/sj.462.0265.

Qian, L., Z. Luo, Y. Du, and L. Guo (2009), Cloud computing: An overview, in *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, Springer-Verlag, Berlin, Heidelberg.

Ryck, P. D., L. Desmet, P. Philippaerts, F. Piessens, and K. U. Leuven (2011), A security analysis of next generation web standards, *Tech. rep.*, ENISA - European Union Agency for Network and Information Security.

Shanmugam, J., and M. Ponnavaikko (2008), Cross Site Scripting-Latest developments and solutions: A survey, *International journal of Open Problems in Computer Science and Mathematics*, *1*(2).

Sridharan, M., S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg (2011), F4f: taint analysis of framework-based web applications, *SIGPLAN Not.*, *46*, 1053–1068, doi:http://doi.acm.org/10.1145/2076021.2048145.

Strein, D., H. Kratz, and W. Lowe (2006), Cross-language program analysis and refactoring, in *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pp. 207–216, IEEE Computer Society, Washington, DC, USA, doi:10.1109/SCAM.2006.10.

Su, Z., and G. Wassermann (2006), The essence of command injection attacks in web applications, *SIGPLAN Not.*, *41*, 372–382, doi: http://doi.acm.org/10.1145/1111320.1111070.

Symantec (2013), Symantec internet security threat report, *Tech. rep.*, Symantec.

Tripp, O., M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman (2009), Taj: effective taint analysis of web applications, *SIGPLAN Not.*, *44*(6), 87–97, doi: 10.1145/1543135.1542486.

Trivero, A. (2008), Abusing html 5 structured client-side storage, *Tech. rep.*, SecDiscover.

West, W., and S. M. Pulimood (2012), Analysis of privacy and security in html5 web storage, *J. Comput. Sci. Coll.*, *27*(3), 80–87.

Zissis, D., and D. Lekkas (2012), Addressing cloud computing security issues, *Future Generation Computer Systems*, *28*(3), 583 – 592, doi: http://dx.doi.org/10.1016/j.future.2010.12.006.