SECURE AND TRUSTWORTHY REMOTE JAVASCRIPT EXECUTION

Carlos Serrão, Diogo Rocha
ISCTE – Instituto Universitário de Lisboa
Ed. ISCTE, Av. das Forças Armadas, 1649-026, Lisboa, Portugal
{carlos.serrao, diogo_antonio_rocha}@iscte.pt

ABSTRACT

Javascript is used more and more as a programming language to develop web applications in order to increase the user experience and application interactivity. Although Javascript is a powerful technology that offers these characteristics, it is also a potential web application attack vector that can be exploited to impact the end-user, since it can be maliciously intercepted and modified. Today, web browsers act as worldwide open windows, executing, on a given user machine (computer, smartphone, tablet or any other), remote code. Therefore, it is important to ensure the trust on the execution of this remote code. This trust should be ensured at the JavaScript remote code producer, during transport and also locally before being executed on the end-user web-browser. In this paper, the authors propose and present a mechanism that allows the secure production and verification of web-applications JavaScript code. The paper also presents a set of tools that were developed to offer JavaScript code protection and ensure its trust at the production stage, but also a proxy-based mechanism that ensures end-users the un-modified nature and source validation of the remote JavaScript code prior to its execution by the end-user browser.

KEYWORDS

web applications, JavaScript, security, trust, proxy

1. INTRODUCTION

For some time now there is a software production paradigm shift, where previously desktop-centric software is migrated to a more distributed and ubiquitous web and mobile-based software (Grove, 2009). Software is currently distributed over the Internet (mostly through the WWW) and accessed and executed on a Web browser. This model consists in fetching code from a remote service (or multiple remote services) and execute that code locally on the end-user web browser – the web application is a client-server software application in which the client-part of the application runs on the web-browser (Segaran, 2007). These web applications, in particular on what concerns the client-side, are mostly based on three different technologies: HTML, CSS and JavaScript. In particular, JavaScript is one of the most important components of a web application, allowing programmers to develop client-side complex logic and interactivity, improving the end-user experience (Flanagan, 2006). Speed on client-side JavaScript execution is one of the most important characteristics of modern web browsers. However, JavaScript is also used as a way for an attacker to compromise a web application. Since JavaScript code is obtained from one or more remote sources and is afterwards executed locally on the end-user browser, it is also possible for an attacker, or even a malicious programmer, to produce or modify the Javascript code prior to its execution, putting in risk the web application itself and consequently the end-user and its own data (Cova, Kruegel, & Vigna, 2010). A major JavaScript attack vector is the non-authorized modification of the code – these modifications can occur at the distributor (server-side), during transport (man-in-the-middle) or even at the destination (man-in-the-browser). Therefore, it is important to ensure JavaScript code trust in all moments, in particular its integrity and origin. JavaScript code can be intercepted, change by an attacker and later executed on the user web browser for malicious purposes, without any warning (Nikiforakis et al., 2012). Moreover, JavaScript code being executed by the browser might not be trustworthy. These attacks authors compromise popular web sites and redirect users to their own malicious

versions (phishing) deceiving users, forcing them to give away private information such as bank account numbers, credit card numbers, personal access codes and much more. Therefore it is important to ensure the security and integrity of the remotely obtained Javascript code in all phases of its existence and execution (since its creation) to enable the appropriate trust mechanisms, protecting the final user (Patil, Dong, Li, Liang, & Jiang, 2011).
The major contribution of this paper is the identification of some of the security challenges associated with the remote Javascript execution and to present a proposal, based on public-key cryptography to create the appropriate mechanisms for protecting web application JavaScript code and also build the necessary trust, integrity and confidentiality mechanisms to ensure the security of the remote JavaScript code before its execution by the browser web. This paper starts by introducing the context of web application and describe their major problems in terms of security. On the following section the major attack vectors to the JavaScript lifecycle are presented and described. The description of the methods that are going to be used to provide the necessary confidentiality and trust characteristics to the JavaScript source-code will be detailed in the next section. After this, the authors present the tools developed to implement the previously mentioned mechanisms and describe its operation. Finally, some conclusions of this work are presented and some future worked directions are pointed out.

2. JAVASCRIPT ATTACK VECTORS

A central component of a web application is JavaScript. JavaScript allows the development of complex logic and advanced interaction mechanisms at the web application client-side. However, this architecture is prone to error and vulnerabilities that can be explored by malicious attackers. Cross-Site Scripting (XSS) attacks and its variants Stored XSS Attacks, Reflected XSS Attacks and DOM Based XSS Attacks consists on the injection of JavaScript code in order to manipulate the logic of the web application, subverting it and allowing an attacker to obtain possible advantages over the web application and the user that is using it (Stuttard & Pinto, 2011). Cross-site Request Forgery (CSRF) is another class of attacks that affect web applications and JavaScript, affecting the end-user forcing him to conduct non-intended operations on the applications it trusts (Barth, Jackson, & Mitchell, 2008). There are multiple threats that affect web applications with particular impact on JavaScript. Having into consideration the web applications and JavaScript lifecycle (Figure 1) it is possible to consider a significative number of attack vectors.

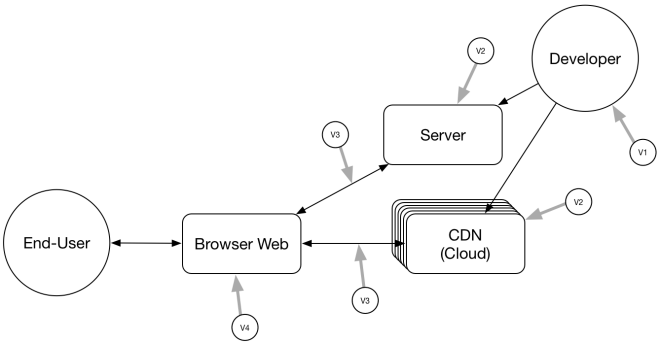

Figure 1. The different attack vectors on the JavaScript lifecycle

This lifecycle covers the development of the web application (and also JavaScript), its storage and distribution, and finally its execution on the client-side web browser. In this lifecycle it is possible to identify a set of attack vectors (V1, V2, V3, and V4) that can be seen as opportunities for attackers to try to exploit the web application and target end-users.

The programmer as an attacker (V1)

In this scenario, the web application developer can be seen as an attacker that produces malicious JavaScript code. This malicious code can be embedded on the web application and executed on the end-user web browser without its own knowledge. This malicious code can be used to obtain details from the end-user environment, mislead the end-user to conduct different non-normal operations, or give more information than it is supposed to. There is also a small variation from this threat that refers to the fact that the developer might be tricked by

a third attacking party to inject malicious JavaScript into legit code. This is, for instance, the case where a developer wants to include some external library into its own code and, accidentally, includes malicious third party code on its own source-code that, ultimately, will be executed at the end-user browser.

Application distributors modify the application (V2)

After being developed, the web application is deployed on a server or Content Distribution Network (CDN) where it can be accessed by multiple users across the World, through a web browser over the Internet. This specific attack vector considers the case where this distributor is also an attacker. The distributor, acting as a malicious attacker (or any third party that was able to subvert the distribution infrastructure) can access the web application source-code (including JavaScript sources) and modify it through the injection of rogue JavaScript. Although improbable, this is also a scenario that needs also to be take into consideration in order to offer an affective protection of the web application source code throughout all the lifecycle stages.

Attacking the communication channel (V3)

Whenever the client requests the web application, the source-code is downloaded from a server before it is executed. While all this source-code is traveling from the server to the client there is the opportunity for an attacker to listen to the communication channel, intercept the code, modify and redirect it to the end-user browser, as if it was the original unmodified code (man-in-the-middle attacks) (Callegati, Cerroni, & Ramilli, 2009). There a large number of tools that can be used to sniff HTTP intercepting the traffic and retrieving communication data. Even an HTTPS ciphered connection can be targeted by these man-in-the-middle attacks – an attacker can setup a rogue proxy that can intercept HTTPS ciphered traffic, decipher it, modify and send it back to the client. The client receives the modified source-code and executes it locally.

Client-side attacks (V4)

Finally, it is also possible to consider an attack vector related with the local execution of JavaScript code, in which malicious software (planted on the web browser or any other compliant user device by an attacker) can act over the legit JavaScript code and inject malicious instructions on it – also known as man-in-the-browser (MITB) attacks (Dougan & Curran, 2012). These attacks use a similar approach to MITM attacks but the user requests interception and modification is conducted by malware that runs between the browser and its security mechanisms, tricking the end-user to believe that everything is absolutely normal.

3. BUILDING TRUST IN JAVASCRIPT

Although JavaScript security mechanisms are already in place, either built-in on the web browser or offered by external plugins or extensions, none of such mechanisms offer end-to-end trust at the JavaScript source-code level. These systems are limited to the defense of the JavaScript code through mechanisms life obfuscation, vulnerabilities identification (JavaScript analyzers) or by blocking the access to non-desirable domains (NoScript browser extension, for instance). However, there are no browser internal or external mechanisms that limit the code execution according to different pre-established conditions nor mechanisms that warrant the origin and integrity of the code since its creation until its execution on the web browser.

End-to-end trust as it is presented in this work refers to the possibility of strongly assuring that the JavaScript being executed by the web browser was originally created by a given authenticated developer and that the code has not been tampered by any external entities. In order to attain these objectives, the authors devised a set o mechanisms implemented through two different tools: “ScriptProtector” and “ScriptProxy”. “ScriptProtector” is the tool used by the developer to create the protection mechanisms that are used to protect and create trust on the produced JavaScript source-code. “ScriptProxy” is the tool used by the end-user that verifies the authenticated code and the code present in the page and validates the browser trust on it.

3.1 Creating and obtaining developer credentials

In order for these two tools to work a set of cryptographic mechanisms need to be setup. The process is based in public-key cryptography and therefore certification authorities (CA) will be used to issue credentials to software development companies and individual programmers. Depending on the trust level, these CA can be public or privately explored by software development companies (SDCA). These CA must have a key-pair $\left(K_{CA}^{pub}, K_{CA}^{priv}\right)$ and a self-signed certificate $\left(Cert_{CA}^{CA}\right)$.

3.1.1 Credentials for individual developers

In this case the developer will get the credentials from a CA in order to be able to produce the code and digitally signed it:

- A developer (SD_1) has a key pair: $K_{SD}^{pub}, K_{SD}^{priv}$
- The developer submits its public key together with other CA requested information: K_{SD}^{pub};
- The CA verifies the information sent by the developer and using its own private key $\left(K_{CA}^{priv}\right)$to issue a digital certificate for the developer $\left(Cert_{SD_1}^{CA}\right)$.

3.1.2 Creating company credentials

In this second situation is the development company that will be certified and afterwards can issue their own credentials to their own developers. As an alternative all the developers on the company will use the same digital certificate.

- In the first scenario we have a CA that issues a digital certificate for a specific company certification authority (SDCA). After submitting its public key $\left(K_{SDCA}^{pub}\right)$the CA issues a certificate for SDCA $\left(Cert_{SDCA}^{CA}\right)$.
- In the second scenario there is only a single certificate that the software development company (SDC) can use globally $\left(Cert_{SDC}^{CA}\right)$.

3.2 Javascript code protection

The developer will need to protect and ensure trust on the produced JavaScript source code. In order to ensure this, the developer will use cryptographic mechanisms that will implement these two requirements. A web application is composed by several components, mostly HTML pages and Javascript scripts that may be included inside or outside an HTML page (Figure 2). The protection mechanism will consider both inline scripts and scripts which are referenced by the HTML page on the web application.

3.2.1 Integrity protecting and trust assurance

In this case, the Javascript scripts will be properly identified the inline scripts $(Script_1, Script_2, \ldots Script_n)$ and also the remote scripts $(RScript_1, RScript_2, \ldots RScript_n)$ that will be protected, through digital signature, either by the individual developers (1) or directly by the software development companies (2).

(1) $DSig_{Script_m}^{SD_n} \rightarrow K_{SD_n}^{priv}\left(Hash_{SHA1}(Script_m)\right), \{n, m \in \mathbb{Z}: 1 \leqslant n, m \leqslant \infty\}$

(2) $DSig_{Script_m}^{C} \rightarrow K_{C}^{priv}\left(Hash_{SHA1}(Script_m)\right), \{m \in \mathbb{Z}: 1 \leqslant n, m \leqslant \infty\}$

Besides that, after all the scripts are properly signed, the full HTML document is also signed in its full extension (1) (2).

(1) $DSig_{HTML}^{SD_n} \rightarrow K_{SD_n}^{priv}\left(Hash_{SHA1}(HTML)\right), \{n \in \mathbb{Z}: 1 \leqslant n, m \leqslant \infty\}$

(2) $DSig_{HTML}^{C} \rightarrow K_{C}^{priv}\left(Hash_{SHA1}(HTML)\right), \{m \in \mathbb{Z}: 1 \leqslant n, m \leqslant \infty\}$

Together with the web page properly protected in terms of integrity and trust the CA (or SDCA) certificate is sent together with the web page to the client. This will allow the client to validate the digital signatures and therefore the integrity and trust on the different scripts.

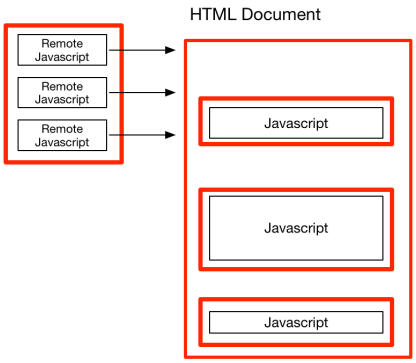

Figure 2. Structure of the HTML document with the elements to protect

3.2.2 Scripts confidentiality protection

In the specific case of the confidentiality protection, it is required to avoid the access of an attacker to the Javascript source-code. Depending on the security policy it is possible to consider the following scenarios:

1. A single key is used to protect all the different scripts on the HTML web page, and it does not change according to the end-user requesting it: $\bigcup_{i=1}^{n} S_k(Script_n)$.
2. Using multiple protection keys, each of the keys is used to protect a single script and do not change according to the end-user requesting it: $\bigcup_{i,n=1}^{j} S_{k_i}(Script_n)$.
3. A variant from the previous two presented scenarios is the one in which the keys are changed according to the end-user. Therefore different S_k are selected and applied for the script protection whenever the HTML page is requested by the end-user.

As a result, from this process, the Javascript source-code will be ciphered with a key that would need to be sent to the end-user – the proxy that will be responsible for the scripts validation prior to its execution by the Web browser.

3.3 Javascript protection execution proxy

In the Javascript execution protection process, there is a proxy that runs on the client-side that receives the content and immediately before passing it to the browser performs a set of validations to verify the remote Javascript trust, integrity and authentication. In order for this proxy to work in a security perspective, the following requirements are necessary:

1. When the proxy is executed for the first time, the proxy (P) creates a key-pair $\left(K_P^{pub}, K_P^{priv}\right)$;
2. The proxy contains on an internal database a list of trustworthy certification authorities (and root certificates) properly setup: $\left(Cert_{CA_1}^{CA_1} \ldots Cert_{CA_n}^{CA_n}\right)$. These certificates are necessary to ensure trust every time signed Javascript is sent to the user Web browser.

3.3.1 Javascript integrity protection and trust verification

This will be the most common usage that will be used for developers that will allow the Javascript integrity and trust. In this situation, both local ($DSig_{Script_m}^{SD_n}$) and remote ($DSig_{RScript_m}^{SD_n}$) scripts are digitally signed and would need to be validated by the proxy before being executed by the web browser or discarded. The verification process is the following:

1. Extraction of the integrated digital certificate that is present on the web page HTML file: $Cert_{SD_n}^{\mathrm{CA}}$.

2. Validate the digital certificate comparing it with the existing proxy trustworthy certification authorities database –additional validations may also be used, such as OCSP (Myers, Ankney, Malpani, Galperin, & Adams, 1999).
3. Finally after the trust is established on the certificate emitting entity is also possible to trust the certificate public-key: K_{SD}^{pub}.
4. This public-key can be used to validate the HTML file digital signature: $VDSig_{HTML}^{SD_n} \rightarrow K_{SD_n}^{pub}\left(DSig_{HTML}^{SD_n}\right), \{n \in \mathbb{Z}: 1 \leqslant n \leqslant \infty\}$.
5. After validating the HTML digital signature its necessary to validate all the other scripts digital signatures:
    a. $VDSig_{Script_m}^{SD_n} \rightarrow K_{SD_n}^{pub}\left(DSig_{Script_m}^{SD_n}\right), \{n, m \in \mathbb{Z}: 1 \leqslant n, m \leqslant \infty\}$
    b. $VDSig_{RScript_m}^{SD_n} \rightarrow K_{SD_n}^{pub}\left(DSig_{RScript_m}^{SD_n}\right), \{n, m \in \mathbb{Z}: 1 \leqslant n, m \leqslant \infty\}$
6. The deciphered scripts, provided by the digital signatures (local and remote) are validated with the respective scripts presents in the page, to ensure that what was signed is the same of what is present in the page.
7. If all the validations were successfully accomplished, the page can be delivered to the web browser for rendering.

3.3.2 Javascript confidentiality protection

This is the additional process that ensures the confidentiality and intellectual property protection of the Javascript source-code. After being assured the integrity and trust on the code on the previous step, the proxy already has the digital certificate of the script producer ($Cert_{SD_n}^{\mathrm{CA}}$) that contains the public-key of the producer ($K_{SD_n}^{\mathrm{pub}}$). With this public-key the proxy will send a new request to the server to get the appropriate key(s) to access the Javascript source-code. The client after validating the answer from the client will select the appropriate secret-key (S_k) and sends this key to the server ciphered ($K_{SD_n}^{\mathrm{pub}}(S_k)$). The software development company deciphers the key ($K_{SD_n}^{\mathrm{priv}}\left(K_{SD_n}^{\mathrm{pub}}(S_k)\right) \rightarrow S_k$). This key is used to protect the different scripts sent from the server to the end-user ($\bigcup_{i=1}^{n} S_k(Script_1)$). The protected scripts are sent to the proxy that uses the appropriate secret-key to decipher them before passing them to the web browser ($\bigcup_{i=1}^{n} S_k(Script_1)$).

4. "SCRYPTPROTECTOR" AND "SCRYPTPROXY"

In order to implement and test the mechanisms proposed and described before two different tools were implemented – "ScriptProtector" and "ScriptProxy". While the "ScriptProtector" was the tool used by the developers to create the trust, integrity and confidentiality required by the Javascript files in the web application, the "ScriptProxy" is the tool used by the different end-users to verify the trust, integrity and protection of those scripts.

4.1 ScriptProtector

The "ScriptProtector" was developed as a command line tool that developers could use to protect and build trust on the local and remote JavaScript source-code required by the web application. This tool starts by parsing the web application files looking for different local JavaScript source-code but also for the different remote JavaScript included by this web application resource (Figure 3). After all the different scripts are identified by the tool it is necessary to apply the necessary trust and integrity protection measures and optionally, if the developer requires so, apply also the confidentiality protection. After all these processes are completed, the final version of the protected web application resource file is outputted to the filesystem. This process is repeated for all the web application resources that need to be protected.

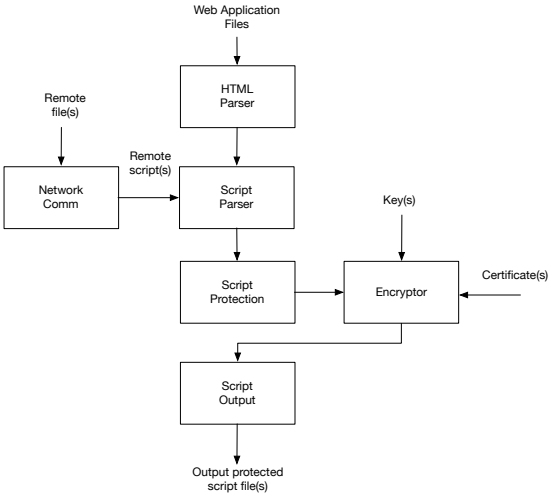

Figure 3. "ScriptProtector" tool architecture

An example of a simple resource to protect (on the left) and the protected version (on the right):

```
<html>
    <head>
        <title>Page Sample for RSA verify in JavaScript</title>

        <script language="JavaScript" type="text/javascript">
            alert("Javascript Signed");
        </script>
    </head>
    <body>
        <h1>Page Sample for RSA verify in JavaScript</h1>
    </body>
</html>
```

```
<html>
 <head>
  <title>Page Sample for RSA verify in JavaScript</title>
  <form name="formSig0" id="formSig0">
   <input type="hidden" name="cert0" id="cert0" value="-----BEGIN CERTIFICATE-----MIIBvTCCASYCCQD55fNzc0WF7
   <input type="hidden" name="siggenerated0" id="siggenerated0" value="ad0d24851db62afcc119b80f326169c22383
   <script language="JavaScript" type="text/javascript" class="js">
    alert("Javascript Signed");
</script>
  </form>
 </head>
 <body>
  <h1>Page Sample for RSA verify in JavaScript</h1>
 </body>
</html>
```

The protected web application resource protected by "ScriptProtector" adds extra information (developer certificate with his public key and the digital signature of each script) on the resource that will enable the establishment of trust and integrity – in this case, confidentiality was not a requirement.

4.2 ScriptProxy

On the end-user side, it was developed a tool that is responsible for assuring the integrity, trust and confidentiality on the JavaScript resources before passing them to the web browser. There were two different choices for the development of such tool – the first would be to develop a specific web-browser extension (or plugin) that would work inside the browser web, while the second would be an independent web-proxy software to which the browser was connected to intercept the web browser requests and server responses. Due to the more platform independent characteristics of the web-proxy, this was the choice selected.

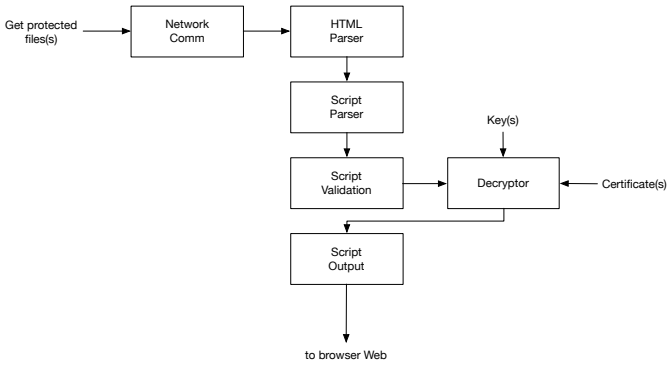

Figure 4. The "ScriptProxy" architecture overview

The web-proxy after intercepting the request from the web-browser, requests the appropriate web application requests from the web-server. After receiving the data, parses it and identifies the protected scripts on the received web-pages resources, and processes them, according to the mechanisms defined previously to ensure the trust, integrity and confidentiality of the Javascript present on the web application resources received. After all the validations are

performed and the scripts are unprotected, the original version of the web application is passed to the web-browser where it can be rendered accordingly (Figure 4).

6. CONCLUSIONS

Web applications are becoming trend applications in our days. The distributed and open nature of the Internet and, in particular, the World Wide Web has made possible the usage of such applications for personal or corporative usage. Critical web applications (such as banking, health and others), that handle personal sensitive data are also becoming more and more frequent and are targeted by attackers that aim specifically at the end-users of such applications.

Exploiting the mechanics of the web application, messing up with their logic, to perform non-authorized actions against their end-users is one of the preferred attack vectors. Most of the times, this is accomplished by tampering the Javascript source-code of the web-application, that is executed locally on the victim's web-browser without any notice.

The system proposed and described in this article presents a set of mechanisms, implemented in two different tools, that allow developers to address the establishment of trust, integrity, confidentiality and intellectual property protection of their own source-code. With it, its possible to create an independent trust bound between the web application producer and the end-user web-browser, to ensure that the source-code executed by the web browser is not tampered with.

Although this is an important step towards making the web applications safer, it is still reduced due to the limitations imposed by the current web-browsers. The level of integration and pre-processing of web content is still limited in most modern web-browsers, thus forcing our tools implementation to be external to the browser itself, having an impact on the end-user experience. In the future it would be desirable to implement the same trust mechanisms either inside the web-browser itself or as a web-browser extension.

REFERENCES

Barth, A., Jackson, C., & Mitchell, J. C. (2008). Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (pp. 75–88).

Callegati, F., Cerroni, W., & Ramilli, M. (2009). Man-in-the-Middle Attack to the HTTPS Protocol. *IEEE Security and Privacy*, *7*(1), 78–81.

Cova, M., Kruegel, C., & Vigna, G. (2010). Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th international conference on World wide web* (pp. 281–290).

Dougan, T., & Curran, K. (2012). Man in the browser attacks. *International Journal of Ambient Computing and Intelligence (IJACI)*, *4*(1), 29–39.

Flanagan, D. (2006). *JavaScript: the definitive guide*. " O'Reilly Media, Inc."

Grove, R. F. (2009). *Web-Based Application Development*. Jones & Bartlett Publishers. Retrieved from http://www.amazon.com/Web-Based-Application-Development-Ralph-Grove/dp/0763759406

Myers, M., Ankney, R., Malpani, A., Galperin, S., & Adams, C. (1999). *X. 509 Internet public key infrastructure online certificate status protocol-OCSP*.

Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., … Vigna, G. (2012). You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 736–747).

Patil, K., Dong, X., Li, X., Liang, Z., & Jiang, X. (2011). Towards fine-grained access control in javascript contexts. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on* (pp. 720–729).

Segaran, T. (2007). *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. "O'Reilly Media, Inc." Retrieved from http://www.google.pt/books?hl=en&lr=&id=7b8N_YCqPH0C&pgis=1

Stuttard, D., & Pinto, M. (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws: Discovering and Exploiting Security Flaws*. John Wiley & Sons. Retrieved from http://www.amazon.co.uk/The-Web-Application-Hackers-Handbook/dp/1118026470