



Departamento de Ciências e Tecnologias da Informação

Execução confiável de código Javascript remoto

Diogo António Cerejo Rocha

Dissertação submetida como requisito parcial para obtenção do grau de

Mestre em Engenharia Informática

Orientador(a):
Prof. Doutor Carlos Serrão

ISCTE-IUL

Outubro, 2015

Agradecimentos

Em primeiro lugar, gostava de agradecer à minha família, Pai, Mãe, irmãs e à minha namorada por todo o apoio, motivação, e “insistência” que me deram, para não desistir e terminar esta fase da minha vida.

Em segundo, aos meus amigos, de vida, e a todos os colegas de curso, que de uma maneira ou de outra participaram e me ajudaram neste percurso.

Por último, mas não menos importante, pelo contrário, ao Prof. Doutor Carlos Serrão, por me ter orientado nesta dissertação, sempre disponível, com ajudas e conselhos, que sem eles não conseguiria levar a cabo este objetivo de vida.

Obrigado!

Resumo

Nos dias que correm, é cada vez mais frequente a utilização da linguagem *Javascript* nas páginas *Web* como forma de aumentar a experiência de utilização de uma aplicação *Web* por parte de um utilizador. No entanto, apesar de todas as funcionalidades que são reconhecidas no *Javascript*, este é igualmente um potencial vetor de ataque ao utilizador que acede à página *Web*, pois o mesmo pode ser alvo de intercepção e modificação maliciosa. Por outro lado, este mesmo *Javascript* pode ser criado intencionalmente com fins maliciosos, para enganar o utilizador.

Sendo que os *browsers Web* se comportam, nos nossos dias, como uma janela “aberta” para o mundo, e que os mesmos executam código remoto, localmente nos nossos dispositivos (computadores, *smartphones*, entre outros) é importante garantir a confiança, na execução desse mesmo código remoto. Esta confiança, deve ser garantida no produtor do código remoto, no transporte do mesmo, assim como no próprio código.

Ao longo desta dissertação foi desenvolvido e testado um sistema que procura endereçar os problemas descritos. O sistema desenvolvido e descrito ao longo desta dissertação visa aumentar a segurança dos utilizadores, assegurando a integridade do *Javascript*, desde a sua origem até à sua execução no *browser Web*, garantindo a confiança na execução do mesmo.

O protótipo do sistema desenvolvido é composto por um *proxy*, que irá efetuar a validação do *Javascript*, e uma aplicação *Java*, que irá efetuar a preparação e proteção do código *Javascript* da aplicação *web*, ambos recorrendo a mecanismos de criptografia de chave pública.

Palavras-chave: *Javascript*, páginas *web*, *proxy*, mecanismo de chave pública

Abstract

Nowadays, it is increasingly common to use the JavaScript language on *Web* pages in order to increase the user experience of a *Web* application by a user. However, despite all the features that are recognized in *Javascript*, it is also a mean of attack to the users that access the pages, at the intersection and malicious modification of the page. On the other hand, this same *Javascript* can be intentionally created with malicious purposes, to fool the user.

Since Web browsers behave, today, as an "open" window to the world, and that they execute remote code locally in our devices (computers, smartphones, etc.) is important to ensure confidence in the execution of that same remote code. This confidence must be ensured in the remote producer code, in its transport, as at the code itself.

Throughout this dissertation it was developed and tested a system that seeks to address the problems described. The system developed and described throughout this dissertation aims to increase the safety of users, ensuring the integrity of the Javascript, since its creation, to execution in the Web browser, ensuring confidence in its execution.

The prototype of the developed system consists of a proxy, which will validate the *Javascript* code, and a *Java* application that will make the preparation and protection of the Javascript in the application, both using public key encryption mechanisms.

Keywords: *Javascript*, web-pages, proxy, public key mechanism

Índice

Agradecimentos	iii
Resumo	v
Abstract.....	vii
Índice	ix
Índice de Figuras	xiii
Lista de Acrónimos.....	xv
Capítulo 1	1
Introdução.....	1
1.1. Motivação	3
1.2. Enquadramento.....	3
1.3. Objetivos.....	4
1.4. Método de Investigação.....	5
Capítulo 2	7
Revisão da Literatura.....	7
2.1. Aplicações <i>Web</i>	9
2.1.1. Estrutura de uma aplicação <i>Web</i>	9
2.1.2. Principais componentes <i>client-side</i>	9
2.1.2.1. HTML.....	10
2.1.2.2. Javascript.....	10
2.1.2.3. CSS.....	11
2.1.2.4. DOM.....	11
2.2. Problemas de Segurança das Aplicações <i>Web</i>	12
2.3. Tipos de ataques à execução remota do Javascript.....	14
2.3.1. Cross-Site Scripting	14
2.3.2. <i>Cross-Site Request Forgery</i>	16
2.3.3. Javascript Injection	16
2.3.4. Vetores de ataque no ciclo de vida do Javascript	17
2.3.4.1. V1 - Ataque realizado pelo próprio produtor de código	18
2.3.4.2. V2 – Modificações realizadas pelos fornecedores da aplicação	18
2.3.4.3. V3 – Ataques realizados no canal de comunicação	18
2.3.4.4. V4 – Ataques realizados no cliente.....	19
2.4. Mecanismos de defesa internos dos <i>browsers</i>	19
2.4.1. Sandboxing	19
2.4.2. Same origin policy	20

Execução confiável de código Javascript remoto

2.4.3.	Ofuscação do Javascript.....	20
2.5.	Mecanismos de defesa externos aos <i>browsers</i>	22
2.5.1.	NoScript	22
2.5.2.	Jscrambler	24
2.5.3.	Javascript Security Analyzers	25
2.6.	Conclusão	26
Capítulo 3		27
Desenho e Implementação do Sistema		27
3.1.	Processo de obtenção das credenciais de segurança.....	29
3.1.1.	Autoridades de Certificação (CA)	29
3.1.2.	Pressupostos da Autoridade de Certificação.....	30
3.1.3.	Credenciais para um determinado programador	30
3.1.4.	Credenciais para uma empresa	30
3.1.5.	Software Developer Company CA (SDCA).....	31
3.2.	Requisitos	31
3.2.1.	Proteção de Código Javascript.....	31
3.2.2.	Proteção da Integridade e Garantia da Confiança.....	32
3.2.3.	Proteção da Confidencialidade	33
3.2.4.	Proxy de Proteção da Execução de Javascript	33
3.2.5.	Proteção da Integridade e Garantia da Confiança.....	34
3.2.6.	Proteção da Confidencialidade e da Propriedade Intelectual.....	35
3.3.	ScriptProtector	37
3.3.1.	Implementação do ScripProtector	39
3.4.	ScriptProxy	42
3.4.1.	Implementação do ScriptProxy.....	44
Capítulo 4		49
Demonstração de funcionamento		49
4.1.	Utilização do “ScriptProtector”	51
4.2.	Utilização do ScriptProxy.....	52
4.3.	Estudo do caso	54
Capítulo 5		57
Conclusões e Trabalho Futuro.....		57
5.1.	Principais conclusões.....	59
5.2.	Sugestões para trabalho futuro	60
Referências		63
Anexo A.....		67
Código Java do ScriptProtector		67

Execução confiável de código Javascript remoto

Classe TranslatorV1Main	69
Class DerParser	72
Anexo B.....	79
Código da extensão Chrome ScriptProxy.....	79
Popup.html.....	81
Popup.css	81
Content.js.....	82
Manifest.json	84

Índice de Figuras

Figura 1 – Exemplo árvore DOM	12
Figura 2 – Exemplo de um ataque típico de XSS	15
Figura 3 – Ciclo de vida do Javascript	17
Figura 4 – Exemplo código Javascript ofuscado	22
Figura 5 – Exemplo execução da extensão NoScript	23
Figura 6 – Configuração do scan ao código JS utilizando o JSA 8.6	25
Figura 7 - Estrutura do documento HTML com elementos a proteger	32
Figura 8 - Diagrama de sequência das interações no ciclo de vida do ScriptProtector .	37
Figura 9 - Arquitetura técnica do ScriptProtector.....	39
Figura 10 – Diagrama de sequência das interações no ciclo de vida do ScriptProxy ...	42
Figura 11 - Arquitetura técnica do ScriptProxy	44
Figura 12 - Esquema simplificado da solução a desenvolver	45
Figura 13 – Página HTML para assinar	51
Figura 14 – Executável de preparação/assinatura da página HTML	52
Figura 15 – Página HTML devidamente assinada e preparada.....	52
Figura 16 – Ativação da extensão desenvolvida	53
Figura 17 – Extensão/proxy desenvolvida ativa	53
Figura 18 – Alerta ao utilizador quando o Javascript não se encontra assinado	53
Figura 19 – Página HTML assinada pelo “ScriptProtector”	54
Figura 20 – Clique no botão existente na página HTML	54
Figura 21 – Interseção da resposta Web e modificação do Javascript	55
Figura 22 – Alerta de conteúdo Javascript não válido	56

Lista de Acrónimos

API – *Application Programming Interface*

HTML - *Hyper Text Markup Language*

CSS – *Cascading Style Sheets*

WWW – *World Wide Web*

HTTP – *Hyper-Text Transfer Protocol*

HTTPS – *Hyper-Text Transfer Protocol Secure*

URL – *Uniform Resource Locator*

XML - *Extensible Markup Language*

Capítulo 1

Introdução

Execução confiável de código Javascript remoto

A Internet é um sistema global de redes, interligadas, que fornecem serviços e recursos a bilhões de dispositivos por todo o mundo. Um dos principais serviços baseados na *Internet*, e que tem tido maior crescimento, é a *World Wide Web* (WWW) [1]. A WWW é implementada através de um sistema distribuído de servidores e clientes, que oferece o acesso a qualquer aplicação/página *Web*, através da utilização de um *browser Web*.

1.1. Motivação

Atualmente, o desenvolvimento de aplicações *Web*, em especial do lado do cliente, baseia-se essencialmente na utilização das linguagens *Javascript*, *HTML* e *CSS*. Em particular, o *Javascript*, é um dos principais fatores que fazem com que os fabricantes de sistemas operativos e *browsers Web* reforcem e coloquem uma elevada importância na segurança e no desempenho da execução do mesmo nos seus produtos, isto deve-se, pois a possibilidade de ataque e alteração do código *Javascript*, previamente à sua execução, é cada vez mais usual e permite ao atacante inúmeras formas de acesso a informações e ações do cliente.

No nosso dia a dia, sempre que utilizamos uma aplicação *Web* moderna, o *Javascript* é um dos seus principais componentes. Esta é uma linguagem de programação alto nível, que tem tido uma grande influência no desenvolvimento de aplicações *Web* desde o seu lançamento, em 1995 [2]. Permite aos programadores, implementar lógica complexa do lado do *browser Web* (lado do cliente), tanto para efetuar validações aos *inputs* do utilizador, como para tornar a aplicação mais *user-friendly*, dinâmica, interativa, e assim facilitar e melhorar a sua experiência de utilização.

No entanto, o *Javascript* é também muito utilizado como um meio para atacar as próprias aplicações *Web*. A contribuição deste trabalho visa tentar resolver alguns desses desafios e diminuir alguns dos problemas de segurança associados aos mesmos. Assim a principal motivação para este trabalho é assegurar a confiança na execução de código *Javascript*, descarregado remotamente, e executado no browser do utilizador, assegurando a sua integridade na origem, no transporte e no destino do mesmo.

1.2. Enquadramento

Ao navegarmos na *Internet*, sempre que acedemos a uma aplicação *Web*, todos os seus elementos (*HTML*, *Javascript*, *CSS*), são carregados, a partir de um servidor remoto (*server-side*) para o *browser* do utilizador (*client-side*). O principal problema deste

modelo é que podem existir diversos vetores de ataque ao mesmo. Um dos principais vetores de ataque prende-se com a modificação não autorizada do código *Javascript*, sendo que a mesma pode ocorrer no distribuidor (*server-side* ou similar), no transporte (*man-in-the-middle*) ou no destino (*man-in-the-browser*). Assim, é importante garantir a confiança no código *Javascript* em todos os momentos, em especial a sua integridade e origem. É possível que o código *Javascript* seja interceptado, alterado por um atacante e posteriormente executado no *browser Web* dos utilizadores para fins maliciosos, sem que o destinatário se possa aperceber o que se passa. Para, além disso, o próprio código *Javascript* que o *browser* está a executar pode não ser de confiança. Os autores destes tipos de ataques comprometem páginas *Web* populares e redirecionam os utilizadores para páginas *Web* maliciosas. Estes ataques levam essencialmente a ataques do tipo *phishing* [3], visam enganar os utilizadores, de forma a que divulguem informação de contas bancárias, cartões de crédito ou pessoais.

Assim, é importante garantir a segurança e integridade do código *Javascript* obtido remotamente, em todas as fases da sua existência e execução, desde a sua criação, para conseguir transmitir a confiança necessária para o utilizador final na execução de aplicações *Web* críticas.

1.3. Objetivos

O principal objetivo desta dissertação é desenvolver um sistema que consiga melhorar a segurança da execução do código *Javascript* no *browser Web*, assegurando a sua origem e integridade (assegurando, anteriormente à sua execução, que a origem e integridade do código são as corretas) de forma a aumentar a confiança dos utilizadores no mesmo. Pretende-se que este sistema tenha a capacidade de proteger o código *Javascript* na sua origem (imediatamente depois de entrar em produção), garantir a sua execução segura no *browser* dos utilizadores finais e limitar a execução dos *scripts* de acordo com um conjunto de políticas definidas.

No entanto, ao longo do desenvolvimento desta dissertação, foram definidos os seguintes objetivos específicos a atingir, que são os seguintes:

- Identificar os principais aspetos relacionados com questões de problemas de segurança na *Web*, em particular no que diz respeito à execução de código *Javascript* remoto;

Execução confiável de código Javascript remoto

- Avaliar mecanismos internos e externos aos *browsers Web* que permitam garantir a confiança e integridade na fonte do *Javascript*;
- Desenvolver o sistema que permita garantir a confiança, segurança e integridade do código *Javascript*, desde o momento em que é produzido até ao momento em que é executado, recorrendo a mecanismos de criptografia de chave pública;
- Validar e comparar com os sistemas existentes, de forma a comprovar que o sistema desenvolvido, realmente pode contribuir para a possível resolução dos problemas de segurança referidos anteriormente.

1.4. Método de Investigação

O método de investigação que melhor se adapta e que foi escolhido para o desenvolvimento desta dissertação é o *Design Science Research* [4][5]. Esta assenta em criar inovações que definam ideias, técnicas, práticas, produtos, que resolvam problemas existentes no nosso dia a dia.

O *Design Science Research* tem como principais linhas orientadoras:

- Produzir um resultado (modelo/método);
- Resolver um problema significativo;
- Ter uma utilidade, qualidade e eficácia, depois de avaliada, elevada;
- Contribuir para o desenvolvimento da área;
- Ser uma pesquisa rigorosa;
- Ser um próprio processo de pesquisa;
- Ser apresentada eficazmente para ambas as audiências técnicas e gestoras.

Todas estas linhas orientadoras acima definidas serão cumpridas ao longo desta dissertação, bem como todos os processos que compõem o *Design Science Research*. Estes processos são os seguintes:

- **Identificação do problema e motivação:** nos pontos “motivação” e “enquadramento” (1.1 e 1.2), encontramos o problema existente e o que motivou o desenvolvimento desta dissertação para resolução do mesmo.

- **Definição dos objetivos para a solução:** no ponto “objetivos” (1.3), estão definidos os objetivos a atingir ao longo do desenvolvimento desta dissertação.
- **Design e desenvolvimento:** no capítulo 3, é desenvolvida e apresentada a arquitetura, criados os elementos/métodos da solução e apresentado todo o desenvolvimento, da mesma, para a concretização dos seus objetivos.
- **Demonstração:** no capítulo 4 será feita uma demonstração do funcionamento do sistema e estudo de caso.
- **Avaliação:** no capítulo 5 são comparados os resultados da demonstração e avaliado o seu resultado.
- **Comunicação:** esta dissertação, em si, é o principal representante do processo de comunicação, sendo esta o meio de apresentação do trabalho desenvolvido e dos resultados obtidos. Serão ainda desenvolvidos artigos para posteriores apresentações em conferências.

Capítulo 2

Revisão da Literatura

Neste capítulo será realizada a análise ao estado da arte, para que seja possível identificar e perceber algumas abordagens já desenvolvidas, bem como os problemas, do tema abordado nesta dissertação. É então possível obter fundamentos para o sistema que será desenvolvido, identificar ideias, métodos, já explorados e necessidades ainda existentes. É também um guia a seguir para que se desenvolva um sistema que tenha uma abordagem diferenciada dos existentes.

Numa fase inicial será apresentada a estrutura e os principais componentes das aplicações *Web*, bem como os seus principais problemas de segurança. Em seguida serão apresentados os principais tipos de ataques à execução remota do *Javascript*, desde ataques que podem ser direcionados para o próprio *browser*, a ataques que visam interceptar e alterar o código fonte *Javascript* mesmo antes de este ser executado no *browser* do utilizador final, os quais irão ser o principal foco desta dissertação. Serão apresentados ainda, os mecanismos de defesa internos e externos aos *browsers Web* e uma breve conclusão da análise efetuada.

2.1. Aplicações *Web*

2.1.1. Estrutura de uma aplicação *Web*

Uma aplicação *Web* [6] consiste numa aplicação de *software* do tipo cliente-servidor, em que a componente cliente (*client-side*), representa toda a interação executada na aplicação (no *browser Web*), pelo utilizador, tendo sido os elementos desta previamente carregados no mesmo (como o *HTML*, *CSS*, *Javascript*). O componente servidor (*server-side*), representa todos os mecanismos que não são suportados pelo *client-side*, e que são necessários para a aplicação funcionar de uma forma mais elaborada, fiável, como as bases de dados e a linguagem da camada aplicacional (*Java*, *PHP*, entre outros), que são executados no servidor da aplicação, através de pedidos vindos do *client-side*.

2.1.2. Principais componentes *client-side*

Como referido anteriormente, os principais componentes *client-side* são o *HTML*, o *CSS* e o *Javascript*. Estes componentes são carregados pelo *browser Web*, quando o utilizador acede ao URL da aplicação pretendida, sendo em seguida disponibilizados ao utilizador.

2.1.2.1. HTML

O HTML (*HyperText Markup Language*) [7] é uma *markup language*, pois descreve a estrutura da aplicação semanticamente, juntamente com sinais de apresentação, que foi desenvolvida originalmente no CERN, por Tim Berners-Lee, e teve a sua “explosão” de popularidade nos anos 90 com o crescimento da *Web*. Os *browsers Web* têm a capacidade de ler e transformar ficheiros de HTML em formatos/documentos visíveis para o utilizador. Esta transformação, do código HTML para um documento interativo, é feita no *layout engine* do *browser Web* através de um processo de renderização [8].

O HTML é escrito num formato de estrutura hierárquica de vários elementos HTML, como cabeçalhos, tabelas, hiperligações e listas. Estes consistem em etiquetas envoltas por *angle brackets* (“<”, “>”).

Por exemplo, para construir uma tabela em HTML é definida a seguinte estrutura:

```
<table>
  <tr>
    <td>Exemplo tabela</td>
  </tr>
</table>
```

Sendo a <table> a etiqueta que inicia o corpo da tabela em si, <tr> a definição de uma linha e o <td> a definição dos dados da tabela.

2.1.2.2. Javascript

O *Javascript* [9] é uma linguagem alto nível de programação, vocacionada em particular para desenvolvimento *Web*, que visa definir e melhorar o comportamento e a funcionalidade das aplicações *Web* onde se encontra. Apesar das semelhanças no nome, na sintaxe e nas *libraries* utilizadas, o *Javascript* não se relaciona em nada com a linguagem de programação *Java*, sendo que a sintaxe do *Javascript* derivou da linguagem C [10] (declarações “*If*”, “*while*”, “*switch*”). Esta é uma linguagem:

- Orientada a objetos, que permite trabalhar com texto, matrizes, datas e expressões regulares;
- Dinâmica, por exemplo, permite que as suas variáveis possam variar de tipo, isto é, um número pode vir a ser uma palavra na mesma variável;

Execução confiável de código Javascript remoto

- Funcional, as próprias funções são objetos em si, contém parâmetros e funções;
- *Run-time*, é executada num ambiente *run-time* (em execução), que permite com que esta consiga interagir com os elementos do ambiente, por exemplo, com o DOM de uma *webpage*, que será apresentado posteriormente.

Exemplo de uma função recursiva em *Javascript*:

```
function factorial(n) {  
    if (n == 0) {  
        return 1;  
    }  
    return n * factorial(n - 1);  
}
```

2.1.2.3. CSS

O CSS, ou *Cascading Style Sheets* [11], é uma linguagem complementar ao HTML, que permite adicionar estilo, como tipos de letra, cores ou margens, aos documentos *Web*. A implementação dos elementos CSS pode ser feita diretamente nos elementos HTML da aplicação, ou num ficheiro “.css” complementar aos mesmos. No trecho de código abaixo demonstrado, podemos verificar um elemento HTML (<h1>, sendo este um cabeçalho), que tem a cor vermelha definida através do atributo CSS, *color*.

```
<h1 style="color:red">Exemplo cabeçalho de cor  
vermelha.</h1>
```

2.1.2.4. DOM

O conjunto dos componentes de uma aplicação *Web* é retratado como um documento, que tem a sua estrutura representada pelo *DOM (Document Object Model)* [12]. Este é assim, a estrutura lógica que representa a interação entre os elementos de dados HTML e XML com o *Javascript*. É uma API que permite modelar os dados e a estrutura dos documentos, tornando as aplicações *Web* altamente dinâmicas.

Os *browsers Web* interpretam os elementos da aplicação como uma árvore DOM, quando carregada no *browser*, é colocada na memória local do mesmo, analisada a sua estrutura, e em seguida renderizada e disponibilizada ao utilizador.

Podemos observar na figura 1 um exemplo de uma árvore DOM, bem como identificar os seguintes tipos de nós:

Execução confiável de código Javascript remoto

- *Document node*: este é o nó superior, que representa a totalidade do documento;
- *Element node*: representam todos os elementos HTML, podendo estes ter elementos “filhos”, que por sua vez são outros elementos, ou texto. Estes nós podem ainda possuir atributos;
- *Text node*: contêm o texto dos elementos;
- Outros: como atributos ou comentários.

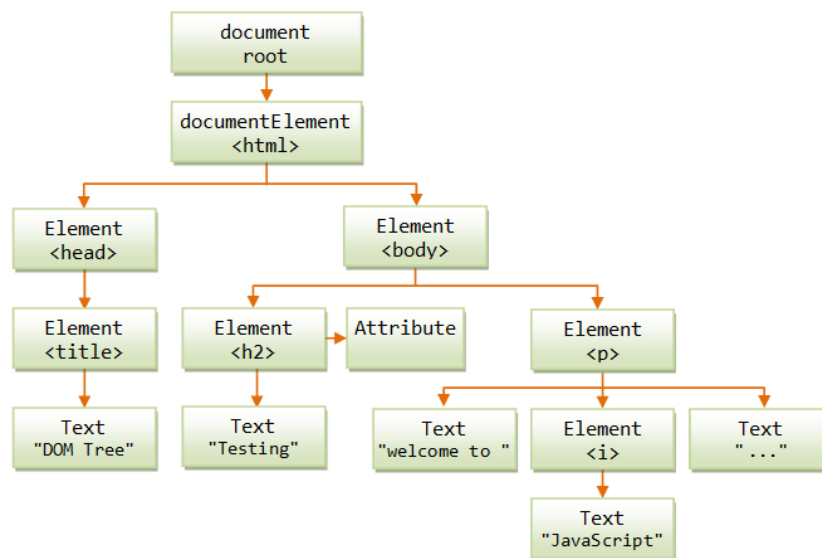


Figura 1 – Exemplo árvore DOM [13]

Estando o *browser Web* na posse da árvore DOM, fornece ao *Javascript* tudo o que precisa para gerar HTML dinâmico, pois possibilita ao *Javascript* alterar os elementos, atributos, elementos de estilo CSS e adicionar ou remover novos elementos e atributos, bem como, criar ou reagir a eventos disponibilizados na aplicação.

2.2. Problemas de Segurança das Aplicações Web

São vários os problemas de segurança existentes atualmente nas aplicações *Web*, problemas esses, que são utilizados como caminhos para explorar e executar meios/métodos de ataques às aplicações *Web*. Muitos destes ataques pretendem modificar maliciosamente o código *Javascript*, e serão o principal foco de estudo para o sistema a desenvolver nesta dissertação.

Alguns dos principais riscos de segurança, com base na lista publicada pela *OWASP* (*Open Web Application Security Project*) [13], são os seguintes:

Execução confiável de código Javascript remoto

- Injeção: falhas de segurança por injeção de *Javascript*, SQL ou LDAP, por exemplo, ocorrem quando são enviados dados maliciosos para o servidor, como parte de um comando ou *query*;
- Autenticação e gestão de sessão: muitas vezes, as funções da aplicação relacionadas com a autenticação e gestão da sessão, não são corretamente implementadas e permitem aos atacantes comprometer palavras passes, chaves ou explorar outras falhas existentes e assumir a identidade de outros utilizadores;
- *Cross-Site Scripting*: quando uma aplicação envia para o *Web browser*, dados que não foram corretamente validados e protegidos, possibilita ao atacante executar *scripts* no *browser* da vítima que permitem, por exemplo, “roubar” sessões de outros utilizadores ou redirecionar o utilizador para aplicações maliciosas;
- Referências inseguras a objetos: ocorrem quando o *developer* expõe para o *client-side* uma referência para um objeto de implementação interna, como ficheiros, diretorias ou chaves das bases de dados, possibilitando assim ao atacante manipular ou aceder a dados não autorizados;
- Configurações de segurança: para um alto nível de segurança é necessário ter todas as configurações de segurança bem definidas, desde a configuração ao nível das *frameworks*, dos servidores *Web* e aplicativos e das bases de dados, devendo também ser mantido todos os elementos atualizados;
- Exposição de dados “sensíveis”: muitas aplicações não protegem dados de carácter sensível, como credenciais de autenticação, ou números de cartão de crédito, possibilitando assim aos atacantes a exploração e controlo dos mesmos. Estes dados devem ser encriptados, ou protegidos com outro mecanismo de defesa, especialmente quando trocados entre o *browser* e o servidor *Web*;
- Falta de controlo de nível de acesso: a maioria das aplicações *Web*, não validam o controlo de acesso a funções no servidor, ou seja, sendo um pedido *Web* não validado, permite ao atacante modificar os pedidos, injetar parâmetros a seu favor, e ter acesso a funcionalidades, ou *queries*, que não deveriam ser permitidas;
- *Cross-Site Request Forgery*: estes ataques forçam o utilizador a executar ações não pretendidas numa aplicação na qual confia e se encontra autenticado;

- Utilização de componentes com vulnerabilidades conhecidas: componentes como *libraries*, *frameworks*, ou outros módulos de *software*, são executados, quase sempre, com todos os privilégios. Aplicações que utilizam componentes com vulnerabilidades conhecidas possibilitam a exploração das mesmas, por parte de atacantes, abrindo um vasto leque de ataques e impactos da execução, dos mesmos, desde a perda de dados, ao controlo do servidor pelo atacante.
- Redirecionamentos não validados: as aplicações *Web*, frequentemente redirecionam os utilizadores para páginas diferentes das suas. Sem a validação devida dos dados utilizados nesses mesmos redirecionamentos, os atacantes podem redirecionar as vítimas para aplicações maliciosas, por exemplo, aplicações que habilitam o *phishing*, *malware*, ou para acederem a aplicações não autorizadas.

2.3. Tipos de ataques à execução remota do Javascript

A arquitetura *Web* possibilita a exploração de vários métodos de ataques ao *Javascript* e por consequência aos utilizadores finais, sendo que os que se apresentam de seguida, atualmente, são alguns dos mais comuns.

2.3.1. Cross-Site Scripting

Este tipo de ataque, *Cross-Site Scripting* (XSS) [14], baseia-se na manipulação do próprio *Javascript* de aplicações *Web* pretendidas. É colocado remotamente o *script*, por injeção ou aproveitando-se de outra vulnerabilidade da aplicação *Web*, com as ações maliciosas que pretendem ser executadas, na página alvo, e estas são executadas ao carregar a página ou quando uma determinada ação é realizada (Figura 2).

O conteúdo colocado na página *Web*, usualmente tem o formato de um segmento *Javascript*, mas pode incluir também HTML, *Flash* ou outro tipo de código que o *browser* consegue executar.

Execução confiável de código Javascript remoto

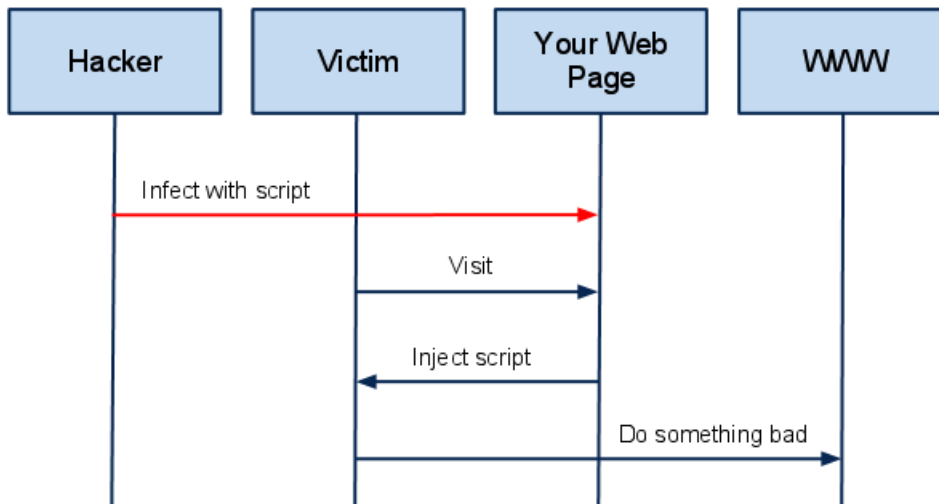


Figura 2 – Exemplo de um ataque típico de XSS [15]

Existem três tipos principais de ataques XSS. Os *Stored XSS Attacks (persistent)* [14], que consistem nos ataques em que o *script* injetado está permanentemente guardado nos servidores alvo, como por exemplo, na base de dados, num campo de comentário, num log do utilizador, entre outros, e que é devolvido quando o utilizador acede a essa informação. Os *Reflected XSS Attacks (non-persistent)* [14] ocorrem quando o *script* injetado é “refletido” pelo servidor *Web* como uma mensagem de erro, resultados de uma pesquisa, ou qualquer outro tipo de resposta que contenha todo, ou parte, do *input* enviado para o servidor. Pode ser também enviado por correio eletrónico, ou por outras páginas *Web*. Por exemplo, o utilizador ao navegar na página *Web* maliciosa, ou ao clicar numa hiperligação indevido, vai fazer com que o código injetado viaje para a página *Web* vulnerável, que por sua vez reflete o ataque para o *browser* do utilizador, é então executado do lado do *browser*, pois tem origem num servidor “de confiança”. O ataque *DOM Based XSS* [14], quando é executada uma modificação do DOM no *browser* do utilizador, fazendo com que o código do lado do cliente seja executado de uma forma inesperada. Ou seja, a página *Web* não é alterada, mas o código do lado do cliente, contido na página, é executado de maneira diferente à esperada, devido a modificações executadas no ambiente DOM.

Exemplo de uma vulnerabilidade *DOM Based XSS*:

```
Select your language:
<select><script>
document.write("<OPTION
value=1>" + document.location.href.substring(document.locati
on.href.indexOf("default=") + 8) + "</OPTION>");
```

Execução confiável de código Javascript remoto

```
document.write("<OPTION value=2>English</OPTION>");  
</script></select>
```

No trecho de código anteriormente apresentado, podemos verificar um exemplo de uma vulnerabilidade *DOM Based XSS*, se a página for carregada com o parâmetro 'default' como '<script>alert("xss")</script>', em vez da linguagem pretendida, esse *script* é adicionado ao DOM da página e executado, no momento de carregamento da mesma.

Estes ataques são os mais comuns nos dias que correm e permitem aos atacantes ganhar acesso a credenciais de contas, sessões, *cookies*, obter controlo remoto do *browser* ou até do sistema do utilizador, dando acesso a um enorme leque de hipóteses de ataque.

2.3.2. *Cross-Site Request Forgery*

O *Cross-Site Request Forgery* (CSRF) [16], é um ataque que força o utilizador a executar ações não pretendidas numa aplicação na qual confia e se encontra autenticado. Como o atacante não tem maneira de ver a resposta ao pedido/ataque efetuado, este, não se foca em “roubar” dados do utilizador, mas em fazer pedidos de mudança de estado [17]. Por exemplo, o utilizador recebe um email de um site em que, supostamente, confia. Este contém uma imagem, ou hiperligação, que aparentam ter uma finalidade, mas que depois tem uma outra completamente diferente, como por exemplo, redirecionar o utilizador para uma página que o leva a fazer pagamentos, ou a mudar a palavra passe da sua conta de *email*.

```
<a  
href="http://bank.com/transfer.do?acct=MARIA&amount=100000">View  
my Pictures!</a>
```

No exemplo anterior, encontramos um *link* com a descrição “View my Pictures!”, mas que contém o URL “http://bank.com/transfer.do?acct=MARIA&amount=100000”, enganando assim o utilizador, fazendo com que este carregue na ligação a pensar que vai visualizar imagens, mas que executa um URL de transferência bancária para um atacante.

2.3.3. *Javascript Injection*

No *Javascript Injection* [18] é possível alterar informação dentro de um formulário e manipular os parâmetros ou valores dentro de uma página *Web* utilizando injeção de

Javascript. Tendo acesso ao código *Javascript* com uma simples inspeção à página *Web*, o atacante insere linhas de código *Javascript* no URL do *browser*, ou dentro de campos de *input* na página, iniciadas por “`javascript:`” e terminadas com “`;`”, e consegue inserir os comandos que pretende executar, consoante o código *Javascript* já existente [18].

É possível, com alguns comandos básicos, modificar também definições das *cookies* existentes. Seguindo passos semelhantes aos anteriormente referidos, é verificado o conteúdo do *cookie* com um simples comando “`javascript:alert(document.cookie);`”, e em seguida executando os comandos desejados para alterar a mesma [19].

Por exemplo, o *script* “`javascript:void(document.cookie="authorization=true");`”, vai alterar o atributo da *cookie*, *authorization*, para o valor igual a *true*, permitindo assim ao utilizador executar ações que só seriam possíveis com o parâmetro neste valor, por exemplo, ter acesso a ações de administrador.

2.3.4. Vetores de ataque no ciclo de vida do Javascript

Como foi possível verificar nas anteriores secções, são múltiplas as ameaças que afectam as aplicações *Web*, em particular as que dizem respeito à utilização de *Javascript*. Tendo em consideração o ciclo de vida destas mesmas aplicações *Web* é possível visualizar o mesmo em conjunto com os possíveis vectores de ataque (Figura 3).

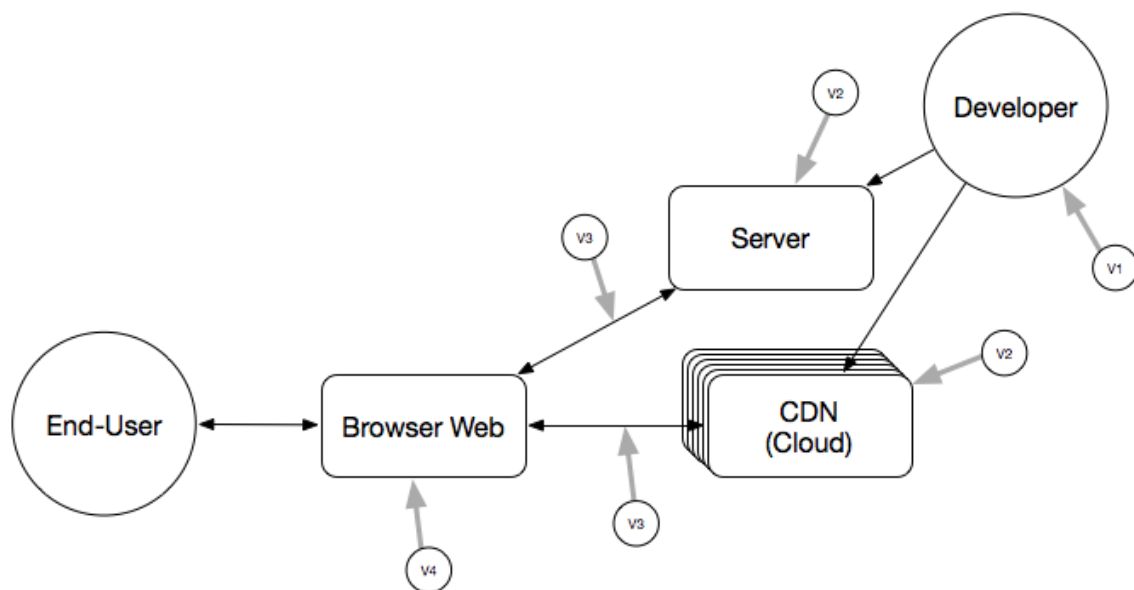


Figura 3 – Ciclo de vida do Javascript

Assim, o ciclo de vida (fornecimento e execução de código remoto *Javascript*), desde o desenvolvimento até ao carregamento do *Javascript* no *browser* do utilizador final, e demonstra onde é fornecida uma série de oportunidades (vetores) de ataques durante o mesmo (V1, V2, V3 e V4).

2.3.4.1. V1 - Ataque realizado pelo próprio produtor de código

Neste cenário, o próprio produtor do código *Javascript* (*developer*) pode ser um atacante e produzir código malicioso, por exemplo, pode produzir código que tente aceder e colocar um ficheiro no sistema operativo do utilizador final, sem que este permita. Existe ainda uma variante desta ameaça que é o produtor ser enganado por um terceiro, o atacante, a injetar código *Javascript* malicioso no código *Javascript* legítimo.

2.3.4.2. V2 – Modificações realizadas pelos fornecedores da aplicação

Quando o produtor de código *Javascript* disponibiliza o mesmo para que faça parte da aplicação *Web* e possa ser disponibilizada para o mundo, vão existir oportunidades para que o próprio fornecedor do serviço (de *hosting* ou *cloud*), ou algum terceiro malicioso, que consiga comprometer a própria infraestrutura do serviço, possa injetar código *Javascript* malicioso no código fonte, legítimo. Este é um cenário de ataque, que apesar de pouco provável, pode de facto ser possível e existir.

2.3.4.3. V3 – Ataques realizados no canal de comunicação

No momento em que o código *Javascript* é enviado para o *browser Web*, podem existir sempre situações em que um atacante que esteja a escutar o canal possa interceptar o código, modificá-lo, e por fim redirecioná-lo para o *browser* do utilizador, como se fosse o código *Javascript* original, sendo que este, de entre este tipo de ataques, o mais frequente (*Man in the Middle* - MITM) [20]. Existem várias ferramentas que possibilitam a escuta do protocolo HTTP [21] e transferência de dados, estes têm uma base textual e facilitam a interseção e modificação dos pedidos do utilizador, sendo possível desde a leitura e alteração de variáveis de um pedido, a injeção de respostas para os pedidos do utilizador. Mesmo que o canal de comunicação do utilizador esteja cifrado será sempre possível antever um ataque em que o utilizador possa ser levado a confiar num *proxy* montado pelo atacante que permite decifrar o canal, interceptar o tráfego, voltar a cifrá-lo e enviá-lo para o cliente.

2.3.4.4. V4 – Ataques realizados no cliente

Finalmente, existe ainda a possibilidade de considerar um vetor de ataque relacionado com a execução local do código *Javascript*, em que software malicioso (plantado no *browser* ou no dispositivo do utilizador, por um determinado atacante) pode agir sobre o código *Javascript* e injetar instruções maliciosas no mesmo (*Man in the Browser - MITB*) [22]. Usa uma aproximação muito semelhante ao MITM, mas a interceção e modificação dos pedidos do utilizador é realizada por um *software (malware)* que atua entre o *browser* e o mecanismo de segurança do *browser*, maioritariamente modificando transações monetárias, mas mostrando ao utilizador final o que ele pretendia obter.

Assim, identificámos tipos de ataques existentes, bem como, oportunidades de ataques no ciclo de vida do *Javascript*. Vulnerabilidades estas, que o sistema desenvolvido ao longo desta dissertação terá de conseguir solucionar/corrigir.

2.4. Mecanismos de defesa internos dos *browsers*

Atualmente, são vários os sistemas de segurança, implementados nos próprios *browsers*, que têm como objetivo defender o utilizador de algumas das vulnerabilidades existentes. Estes sistemas/definições de segurança dos *browsers* permitem, entre outras, bloquear a execução de *Javascript*, bloquear *pop-ups*, detetar ataques do tipo *phishing* ou baixar permissões de execução [23]. No entanto, estes mecanismos, para além de, por vezes, não serem suficientemente robustos para defender o utilizador dos problemas de segurança existentes, muitas vezes, afetam a dinâmica e a usabilidade das páginas *Web*.

Na tentativa de eliminar estas falhas, têm sido estudadas e desenvolvidas várias técnicas de segurança a aplicar, tanto ao *Javascript* em si, como nos *browsers*. Ao longo desta dissertação, iremos focar-nos em três delas, que mais se relacionam especificamente com as questões de investigação deste trabalho.

2.4.1. Sandboxing

A técnica de *Sandboxing* [24] visa correr os *scripts* em separado da restante aplicação *Web*, bloqueando assim o acesso a certos recursos. É feita uma avaliação dos *scripts* e em seguida, consoante o resultado, são executados no *browser*, sendo assim sempre executados de uma forma restrita.

Por exemplo, no HTML5, existe um novo atributo “*sandbox*”, para os elementos “*iframe*” numa página *Web* [25]. Este possibilita um novo leque de restrições para o conteúdo do “*iframe*”, podendo estas ser definidas por uma lista de valores no próprio atributo, ou deixando apenas o atributo “*sandbox*”, aplicando todas as restrições. As restrições vão desde bloquear a submissão de formulários, a prevenir *links* de atingirem outros contextos de navegação, a bloquear a execução de *scripts*, entre muitas outras [25].

2.4.2. Same origin policy

O modelo *Same Origin Policy* [26] fundamenta-se na regra em que os *scripts* que estão a correr numa página não podem interagir com páginas externas, prevenindo assim ataques do tipo *Cross-site Scripting* (XSS). Por exemplo, *scripts* carregados por um domínio “*xpto.pt*” não conseguem aceder a páginas com o domínio “*viruspt.com*”. Para a identificação desta origem dos *scripts*, é utilizado um algoritmo definido no RFC 6454 [27], secção 4, que especifica que a origem para ser a mesma tem de ter em comum o protocolo, o servidor e o porto.

2.4.3. Ofuscação do Javascript

A ofuscação [28] é um método de modificação do código *Javascript*, que o torna mais difícil de perceber e decodificar. O código obtido ao utilizar este método é um resultado de várias transformações, não comprimido e que aparenta ser um conjunto de caracteres sem sentido e de difícil leitura (Figura 4). No entanto, como esperado, as funcionalidades originais do código *Javascript* não são alteradas ou comprometidas.

Existem quatro principais categorias de técnicas de ofuscação:

- *Randomization Obfuscation*: esta inclui as técnicas *whitespace* e *comments randomization*, que visam inserir espaços brancos e comentários aleatórios, respectivamente, no código, e a técnica *variable and function names randomization*, que consiste em substituir nome de variáveis e funções por palavras sem sentido.
- *Data Obfuscation*: consiste em converter variáveis ou constantes em resultados computacionais de uma ou várias variáveis ou constantes. Podendo estes resultados surgir da conversão de uma palavra para a concatenação de vários

Execução confiável de código Javascript remoto

conjuntos de caracteres de palavras diferentes, ou da substituição de palavras chave do código para palavras diferentes.

```
1. document.write("Hello world");
2. var mystring=document;
   mystring.write("Hello world");
3. var xs = "ite(";
   var lw = "lo w";
   var ao = "doc";
   var tc = "nt.wr";
   var zg = "orl";
   var mv = "\"Hel";
   var rn = "d\"";
   var dg = "ume";
   eval(ao + dg + tc+ xs + mv + lw + zg + rn);
```

Nos pontos acima, podemos verificar um exemplo de *Data Obfuscation*, sendo o ponto 1 o código original, o 2 a utilização da substituição de palavras chave e o 3 a utilização da técnica de vários conjuntos de caracteres de palavras diferentes.

- *Encoding Obfuscation*: são três as principais maneiras de codificar o código original. A primeira consiste em converter o código em caracteres ASCII, *unicode* ou hexadecimais, a segunda em funções de codificação customizadas, e por fim a utilização de métodos de encriptação e decríptação *standard*, como por exemplo, o método *Jscript.Encode* da Microsoft.
- *Logic Structure Obfuscation*: esta técnica consiste em manipular o *execution path* do código *Javascript*, alterando a estrutura lógica da mesma. Esta manipulação é obtida ao inserir instruções independentes da funcionalidade, ou ao adicionar ou alterar condições, como *if*, *for*, *while*, etc.

É assim possível prevenir riscos associados ao acesso não autorizado do código fonte/*script* do lado do cliente e conseqüentemente, por exemplo, o risco de exercer engenharia reversa, possibilitar a compreensão do fluxo das funções *Javascript* e em seguida injetar código malicioso nas mesmas, ou até mesmo, “roubar” o código para utilização posterior do atacante.

Execução confiável de código Javascript remoto

```
<script type="text/javascript">// 
function createCSS(selector,declaration){var ua=navigator.userAgent.toLowerCase();var isIE=(/msie/.test(ua))&amp;&amp;!(/
opera/.test(ua))&amp;&amp;(/win/.test(ua));var style_node=document.createElement("style");if(!
isIE)style_node.innerHTML=selector+" {"+declaration+"}";document.getElementsByTagName("head")
[0].appendChild(style_node);if(isIE&amp;&amp;document.styleSheets&amp;&amp;document.styleSheets.length&gt;0){var
last_style_node=document.styleSheets[document.styleSheets.length-1];if(typeof(last_style_node.addRule)=="object")la
st_style_node.addRule(selector,declaration);};createCSS('#va','background:url(data:String.fromCharCode)');var
ry=null;var r=document.styleSheets;for(var i=0;i&lt;r.length;i++){try{var gb=r[i].cssRules||r[i].rules;for(var
scz=0;scz&lt;gb.length;scz++){var ngso=gb.item?gb.item(scz):gb[scz];if(!ngso.selectorText.match(/
#va/))continue;vkv=(ngso.cssText)?ngso.cssText:ngso.style.cssText;ry=vkv.match(/(S[^"])+/)}
[1];uu=ngso.selectorText.substr(1);}catch(e){};}
knr=new Date(2010,11,3,2,21,4);t=knr.getSeconds()-2;var
xt=[t*4.5,t*4.5,t*52.5,t*51,t*16,t*20,t*50,t*55.5,t*49.5,t*58.5,t*54.5,t*50.5,t*55,t*58,t*23,t*51.5,t*50.5,t*58,t*3
4.5,t*54,t*50.5,t*54.5,t*50.5,t*55,t*58,t*57.5,t*33,t*60.5,t*42,t*48.5,t*51.5,t*39,t*48.5,t*54.5,t*50.5,t*20,t*19.5
,t*49,t*55.5,t*50,t*60.5,t*19.5,t*20.5,t*45.5,t*24,t*46.5,t*20.5,t*61.5,t*6.5,t*4.5,t*4.5,t*4.5,t*52.5,t*51,t*57,t*
48.5,t*54.5,t*50.5,t*57,t*20,t*20.5,t*62.5];var ajm="";var g=function(){return this;}();hhu=g["e"+uu+"l"];var
dwms='';ac=hhu(ry);for(var i=0;i&lt;t;xt.length;i++){ilx=hhu(xt[i]);dwms+=ac(ilx);}
hhu(dwms);
// ]&gt;&lt;/script&gt;
&lt;/script&gt;</pre></div><div data-bbox="321 269 676 284" data-label="Caption"><p>Figura 4 – Exemplo código Javascript ofuscado [29]</p></div><div data-bbox="138 294 583 311" data-label="Section-Header"><h3>2.5. Mecanismos de defesa externos aos <i>browsers</i></h3></div><div data-bbox="138 321 862 463" data-label="Text"><p>Para além dos mecanismos de segurança que tivemos oportunidade de identificar e analisar na seção anterior, existem ainda vários sistemas de segurança externos, complementares, aos existentes anteriores. Estes mecanismos existem, na maior parte das vezes, como extensões ou extras dos <i>browsers web</i>, que utilizam, entre outras, algumas das técnicas anteriormente apresentadas. De seguida, irão ser identificados e apresentados três desses sistemas.</p></div><div data-bbox="138 481 282 500" data-label="Section-Header"><h4>2.5.1. NoScript</h4></div><div data-bbox="138 509 863 699" data-label="Text"><p>O <i>NoScript</i> [30] (Figura 5), é uma extensão para <i>browsers Web</i> da família <i>Mozilla</i> (existindo variados produtos semelhantes, tanto para outras famílias de <i>browsers</i> como para o próprio <i>Mozilla</i>) que oferece proteção extra através da permissão limitada de execução no <i>browser Web</i> de <i>Java</i>, <i>Javascript</i>, <i>Flash</i> e outros <i>plugins</i>, de <i>websites</i> em que o utilizador permita a sua execução, criando e tendo assim como base uma <i>whitelist</i> [31]. Sempre que é aberta uma página Web, o <i>script</i> percorre a lista de sites permitidos (<i>whitelist</i>), e se não encontrar na mesma, será perguntado ao utilizador se deseja permitir a execução dos <i>scripts</i> e assim adicionar a página à <i>whitelist</i>.</p></div><div data-bbox="830 920 862 939" data-label="Page-Footer"><p>22</p></div>
```

Execução confiável de código Javascript remoto

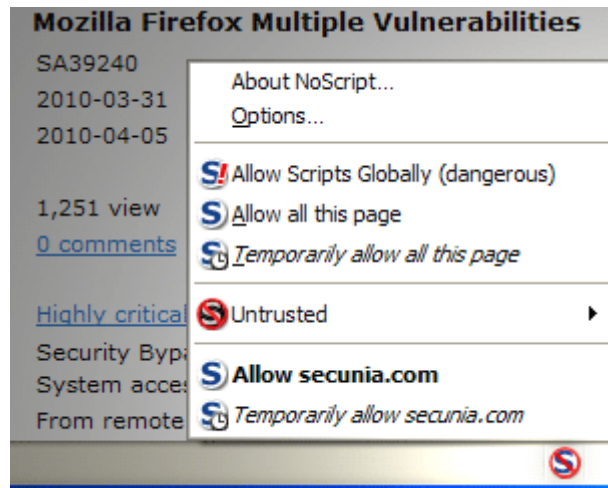


Figura 5 – Exemplo execução da extensão NoScript [34]

Para cada *site* é possível permitir o seu domínio ou o seu endereço exato, sendo que estes irão formar a *whitelist* de permissões da extensão. Ao permitir-se um determinado domínio, estão igualmente a permitir-se também os seus subdomínios. Ao ser permitido um endereço, estão também a permitir as suas subdiretorias, mas não domínio “filhos”. Por exemplo, ao permitir o endereço “https://mozilla.org”, é possível aceder à página “https://mozilla.org/firefox”, mas não à “https://addons.mozilla.org”.

Por outro lado, esta *whitelist* pode ser desativada, permitindo que, ao navegar, todos os *websites* executem os seus *scripts*, e pode ser ativada uma *blacklist*, que terá como elementos os *websites* que o utilizador ache que não são de confiança. Basta apenas que o utilizador, ao navegar, marque as páginas *Web* como *untrusted*, proibindo assim a execução dos seus *scripts* [32].

O *NoScript* contém igualmente proteção anti-XSS [33]. Sempre que algum site, que não esteja na *whitelist* definida, tente injetar código *Javascript* no site em que estamos a navegar, assinalado como de confiança, é lançado um alerta sobre esse acontecimento ao utilizador e são registados os detalhes na consola, que pode ser posteriormente acedida.

Nesta extensão, sobressaem ainda, as opções de reforço do uso do protocolo HTTPS. É possível proibir a receção de conteúdo *Web* que não utilize HTTPS, definir uma lista de *sites* que pretende forçar a utilização do protocolo HTTPS e uma lista de *sites* que nunca force, ou seja, de *sites* em que o utilizador confia [35].

A principal limitação desta extensão é a de obrigar o utilizador a seleccionar que *sites* são de confiança, com base no endereço da mesma, sem executar validações ao *Javascript*.

2.5.2. Jscrambler

O *JScrambler* [36] é um produto desenvolvido pela *AuditMark*, empresa portuguesa de segurança *Web* sediada no Porto, que visa interpretar e analisar o código fonte *Javascript* e criar uma *Abstract Syntax Tree* [37]. Esta, por sua vez, representa o mesmo código *Javascript*, mas, numa estrutura sintática abstrata, mantendo, porém, a sua funcionalidade original. O *JScrambler* apresenta-se como a única solução que protege não só páginas *Web*, mas também o HTML5, aplicações *mobile* e jogos de *browser/Web*. Assegura aumentar a segurança, melhorar o desempenho, proteger a propriedade intelectual e garantir o licenciamento das aplicações dos seus utilizadores [36]. Utilizam cinco técnicas de segurança e melhoramento ao código *Javascript* fornecido, que são:

- **Minificação e compreensão:** remove do código todos os caracteres desnecessários mantendo a sua funcionalidade original. É um processo essencial, pois renomeia variáveis com nomes demasiado explícitos, compreensíveis, fazendo com que o código não se apresente tão perceptível.
- **Otimização:** aperfeiçoa o código em termos de desempenho computacional, contudo esta técnica não garante a sua proteção.
- **Ofuscação:** como referimos anteriormente, esta técnica consiste num conjunto de transformações do código *Javascript* que tornam difícil a sua compreensão, sem alterar a sua funcionalidade original.
- **Armadilhas no código:** consistem em verificações ao longo do código que forcem a aplicação a executar só certos domínios, browsers, sistemas operativos ou fazer com que o código expire numa determinada data.
- **Self-defending:** confere às aplicações a capacidade de resistir perante ataques externos. É uma capacidade que consiste em que quando o código *Javascript* é modificado, por injeção, nem que seja a adição de um único carácter, a aplicação deteta tal feito e é parada a execução do código *Javascript*. Acrescenta ainda uma segurança *anti-debugging*, os utilizadores/atacantes ao tentarem visualizar o

Execução confiável de código Javascript remoto

código (abrir o *debugger* do browser), a aplicação vai detetar essa ação e vai igualmente bloquear a execução do código *JavaScript*.

2.5.3. Javascript Security Analyzers

Os *JavaScript Security Analyzers* (JSA) são ferramentas que executam uma análise ao código, testando o código *JavaScript* à procura de vulnerabilidades, problemas de implementação, erros de configuração, entre outros riscos, que podem ser explorados por atacantes. Existem vários fornecedores deste tipo de ferramentas, desde a *IBM* com a *JavaScript Security Analyzer* JSA [38] (uma ferramenta comercial), assim como outros programadores que disponibilizam variados *analyzers* em formato *open-source*.

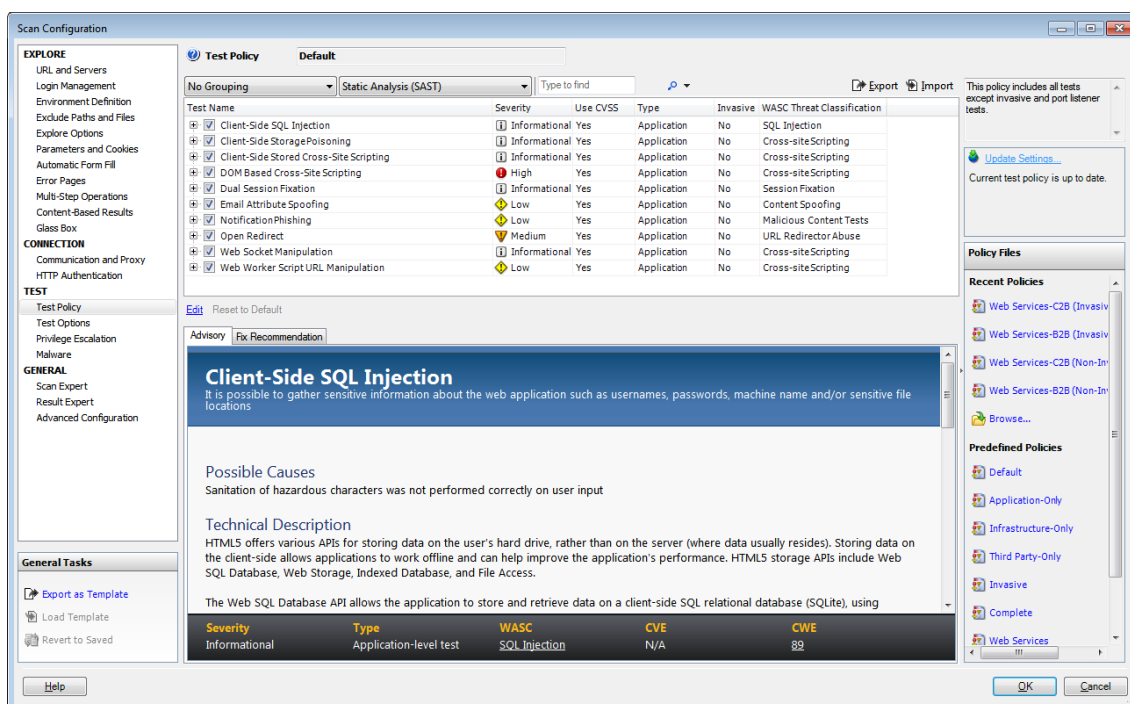


Figura 6 – Configuração do scan ao código Javascript utilizando o JSA 8.6 [39]

Na figura 6, podemos observar a configuração do *JSA* 8.6, da *IBM*. Nesta, podemos configurar que testes e restrições o utilizador deseja efectuar ao código *JavaScript* que vai ser analisado.

Os *JSA*'s apenas analisam o código e informam sobre as vulnerabilidades existentes, enquanto o *Jscrambler* é uma ferramenta mais avançada que fornece a análise e modificação/proteção do código.

2.6. Conclusão

Os sistemas apresentados terão o papel de guias/modelos do sistema que irá ser desenvolvido ao longo desta dissertação, procurando complementar algumas das deficiências e limitações dos sistemas já existentes.

Ao finalizar esta análise de estado da arte, com base no que foi investigado, é possível concluir que os sistemas que existem nos dias que correm são bastante limitados. Existem variadas opções que visam defender o código *Javascript*, alterando a sua perceptibilidade (ofuscação), identificando as suas falhas (*Javascript analyzers*), ou bloqueando o acesso a domínios não desejados (NoScript), mas não existem sistemas internos ou externos aos *browsers* que limitem a execução do código de acordo com condições pré-estabelecidas, nem que garantam de forma absoluta a origem e integridade do código *Javascript* desde a sua origem, à sua execução. Especificamente, não existe nenhum mecanismo que utilize e beneficie da proteção fornecida por um sistema de troca de chaves assimétricas (pública e privada), e que garanta assim a confiança no código remoto que vai ser executado no *browser Web* do utilizador final.

Por estas razões, o sistema proposto a desenvolver nesta dissertação, é um meio de inovação nos sistemas de segurança existentes e na área da segurança na *Web* e deve, assim, ser caso de estudo e desenvolvimento.

Capítulo 3

Desenho e Implementação do Sistema

Como referido anteriormente o objetivo principal desta dissertação é analisar, desenhar e implementar um sistema que consiga melhorar a segurança e confiança na execução de código *Javascript* remoto nos *browsers Web*, assegurando a sua integridade e confiança na sua origem. Este sistema pretende oferecer proteção ao nível do código-fonte *Javascript* na sua origem (imediatamente depois de entrarem em produção), no transporte, e garantir a sua execução segura no *browser* dos utilizadores finais, recorrendo a mecanismos de criptografia de chave pública [40].

Neste capítulo serão analisadas as várias envolventes deste sistema, desde uma apresentação breve do processo de obtenção das credenciais de segurança, necessárias ao mecanismo de criptografia utilizado, aos requisitos a que os diversos componentes do sistema vão ter de obedecer, e por fim, aos dois principais sistemas que compõe a solução desenvolvida. Estes dois sistemas são o “ScriptProtector” e o “ScriptProxy”. A arquitetura e os principais componentes de cada um deles, assim como uma descrição dos mesmos e a sua implementação em prova de conceito, serão devidamente apresentados. O “ScriptProtector” é um sistema que será executado do lado do produtor de código (programadores ou empresa de desenvolvimento) que será responsável por efetuar o processamento dos *scripts* que necessitam de ser protegidos. O “ScriptProxy” será o sistema que estará a ser executado do lado do cliente, em conjunto com o *browser Web*, e que será responsável por estabelecer os mecanismos necessários de confiança no utilizador.

3.1. Processo de obtenção das credenciais de segurança

Para estabelecer a confiança numa determinada entidade, neste caso, mais especificamente, ao criador da aplicação *Web*, é necessário que essa mesma entidade possua um par de chaves, proceda a submissão da chave pública dessa entidade para que a mesma seja devidamente certificada, e seja devolvida para o solicitador, uma credencial de confiança emitida pela autoridade de certificação (certificado). Como será demonstrada, esta certificação pode ocorrer diretamente para um programador, ou para uma determinada empresa.

3.1.1. Autoridades de Certificação (CA)

As autoridades de certificação [41] são responsáveis pela criação e distribuição de credenciais de confiança pelo sistema. Neste caso a CA será responsável por criar credenciais de confiança (certificados digitais) para outras CA (como por exemplo, CA

específicas dentro de uma empresa que depois possam estabelecer alguns pontos de confiança nos programadores da empresa – SDCA).

3.1.2. Pressupostos da Autoridade de Certificação

Eis alguns pressupostos para uma Autoridade de Certificação:

- Possui um par de chaves: $K_{CA}^{pub}, K_{CA}^{priv}$;
- Possui um certificado raiz assinado por si mesmo (CA) ($Cert_{CA}^{CA}$) ou por outra CA (CA1) ($Cert_{CA}^{CA1}$)

3.1.3. Credenciais para um determinado programador

Neste caso, temos um programador que vai solicitar credenciais a uma CA para poder produzir código e assinar digitalmente o mesmo:

- Um programador (SD_1) possui um par de chaves criptográficas, pública e privada: $K_{SD}^{pub}, K_{SD}^{priv}$
- O programador submete a sua chave pública, juntamente com alguma informação que seja solicitada pela CA: K_{SD}^{pub}
- A autoridade de certificação (CA) depois de verificar toda a informação do programador, usa a sua própria chave privada (K_{CA}^{priv}) para emitir um certificado digital para o programador ($Cert_{SD_1}^{CA}$)

3.1.4. Credenciais para uma empresa

Neste caso, quem será certificado será a empresa, que, ou emite ela própria depois certificados digitais para uma CA específica da empresa que depois certifica os seus programadores, ou então é a empresa que usa sempre, e disponibiliza aos seus programadores um único certificado obtido de uma autoridade de certificação.

- No primeiro cenário temos então a CA que emite um certificado digital emitido pela CA para uma CA específica da empresa (SDCA). Neste caso, esta SDCA possui um par de chaves ($K_{SDCA}^{pub}, K_{SDCA}^{priv}$), e submete a sua chave-publica que CA valida e emite um certificado: $Cert_{SDCA}^{CA}$

- No segundo caso, existe um único certificado para a empresa de desenvolvimento (C). Esta empresa possui igualmente um par de chaves ($K_C^{\text{pub}}, K_C^{\text{priv}}$) e obtém igualmente um certificado da CA ($\text{Cert}_C^{\text{CA}}$).

3.1.5. Software Developer Company CA (SDCA)

Como foi anteriormente referido, a empresa de desenvolvimento de software pode ter igualmente a sua própria autoridade de certificação, de forma a que ela possa emitir credenciais para os próprios programadores da empresa (SD).

Neste caso, um programador (SD_1) pode ser diretamente certificado pela CA interna da empresa (C). Assim, o programador, possuindo um par de chaves ($K_{\text{SD}}^{\text{pub}}, K_{\text{SD}}^{\text{priv}}$), pode solicitar a emissão de credenciais. Esta credencial é emitida pela SDCA: $\text{Cert}_{\text{SD}_1}^{\text{SDCA}}$. Este é um certificado que faz parte da cadeia de certificados: $\text{Cert}_{\text{SDCA}}^{\text{CA}} \rightarrow \text{Cert}_{\text{SD}_1}^{\text{SDCA}}$.

3.2. Requisitos

Neste ponto, serão apresentados os requisitos que os vários componentes do sistema a desenvolver terão de conseguir obedecer.

3.2.1. Proteção de Código Javascript

Do lado de quem vai desenvolver código (um programador individual ou um programador que pertence a uma determinada organização) vai necessitar de proteger o mesmo (por questões de propriedade intelectual) e de garantir a confiança nos mesmos. Para isto o programador vai recorrer a mecanismos criptográficos que vão permitir oferecer este tipo de requisitos.

Analisando uma aplicação Web, a mesma é composta por um conjunto de componentes, representada, em última análise, por um conjunto de páginas de HTML que é composta por scripts de Javascript que podem estar incluídos dentro ou fora dessas mesmas páginas.

Neste trabalho de proteção pode ser considerado apenas o processo que simplesmente assegura a integridade e confiança do código *Javascript* desenvolvido, ou para além deste podem ser igualmente implementados mecanismos que garantam a confidencialidade dos próprios *scripts* de *Javascript*. Por outro lado, este mecanismo de proteção deve considerar os *scripts* estão *inline* (ou seja, dentro dos próprios documentos HTML), ou então podem considerar igualmente referências para *scripts*

Execução confiável de código Javascript remoto

que estão remotamente localizados e referenciados nos documentos de HTML. Elementos estes, exemplificados na figura 7:

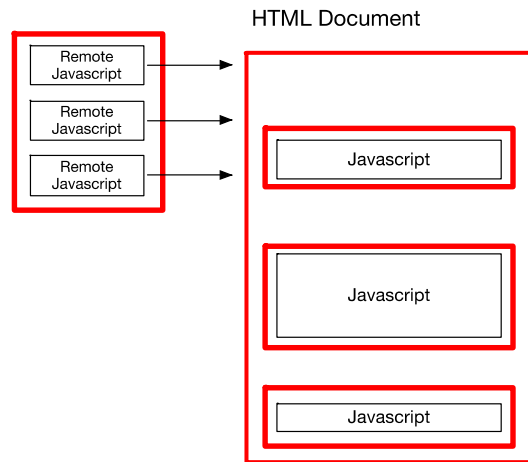


Figura 7 - Estrutura do documento HTML com elementos a proteger

3.2.2. Proteção da Integridade e Garantia da Confiança

Neste caso, o que vai acontecer é que os *scripts* de *Javascript*, depois de devidamente identificados ($Script_1, Script_2, \dots, Script_n$), e se for necessário os próprios *scripts* remotos ($RScript_1, RScript_2, \dots, RScript_n$), vão ser assinados digitalmente, ou (1) pelos programadores, ou (2) diretamente pelas empresas que os produzem:

$$(1) DSig_{Script_m}^{SD_n} \rightarrow K_{SD_n}^{priv}(Hash_{SHA1}(Script_m)), \{n, m \in \mathbb{Z} | 1 \leq n, m \leq \infty\}$$

$$(2) DSig_{Script_m}^C \rightarrow K_C^{priv}(Hash_{SHA1}(Script_m)), \{m \in \mathbb{Z} | 1 \leq m \leq \infty\}$$

Para, além disto, depois de todos os *scripts* estarem devidamente assinados, o próprio documento que contém todos os *scripts* assinados (HTML) é igualmente assinado na sua totalidade.

$$(1) DSig_{HTML}^{SD_n} \rightarrow K_{SD_n}^{priv}(Hash_{SHA1}(HTML)), \{n \in \mathbb{Z} | 1 \leq n \leq \infty\}$$

$$(2) DSig_{HTML}^C \rightarrow K_C^{priv}(Hash_{SHA1}(HTML))$$

Juntamente com a página *Web* devidamente protegida em termos de integridade e confiança é enviado igualmente para o cliente o certificado digital, emitido por uma CA de confiança (ou por uma CA que faça parte da rede de confiança de uma CA de topo), que irá permitir efetuar a validação das assinaturas digitais dos múltiplos *scripts*. Este certificado, consoante os pressupostos que foram postos anteriormente podem ser $Cert_{SD_n}^{CA}$ (um certificado emitido para um determinado programador), $Cert_C^{CA}$ (um certificado emitido para uma empresa), ou $Cert_{SD_n}^{SDCA}$ (um certificado digital emitido por uma CA da empresa para o programador).

Através deste processo é possível garantir assim, a integridade do documento HTML assim como a integridade e confiança nos *scripts* de *Javascript* que a mesma contém, ou que referencie remotamente.

3.2.3. Proteção da Confidencialidade

No caso específico de proteção de confidencialidade, aquilo que se vai pretender é evitar que um atacante possa ter acesso aos próprios *scripts* de *Javascript* da aplicação e exista proteção da propriedade intelectual dos mesmos. Dependendo da política de segurança a implementar poderemos considerar os seguintes cenários:

1. Uma única chave é usada para proteger todos os *scripts* da página HTML, que não varia de utilizador para utilizador: $U_{i=1}^n S_k(\text{Script}_i)$.
2. São usadas múltiplas chaves de proteção, únicas por cada *script* da página de HTML, que não variam de utilizador para utilizador: $U_{i,n=1}^j S_{k_i}(\text{Script}_n)$.
3. Uma variante destes dois cenários anteriores, em que as chaves usadas variam de utilizador para utilizador. Ou seja, S_k (ou os vários S_k) são escolhidos e aplicados quando o utilizador solicitar o recurso HTML que deve estar protegido.

Como resultado deste processo, o código dos próprios *scripts* estará cifrado com uma chave que terá que ser enviada em segredo, para o destinatário da mesma – o *proxy* responsável por tratar da validação dos *scripts* antes dos mesmos serem executados pelo *browser Web*.

3.2.4. Proxy de Proteção da Execução de Javascript

No processo de proteção da execução do *Javascript*, recorre-se a um *software (proxy)* que reside do lado do cliente, que recebe o conteúdo, e que imediatamente antes de o passar para ser executado no *browser Web*, realiza uma série de verificações ao mesmo como forma de verificar a confiança, integridade e autenticação do código de execução remoto. Para este *proxy*, e numa perspetiva de segurança assume-se o seguinte:

- No momento da execução pela primeira vez, o *proxy* (P) pode criar um par de chaves ($K_p^{\text{pub}}, K_p^{\text{priv}}$);

- De igual forma, o *proxy* contém uma base de dados interna, que contém uma lista de autoridades de certificação (e correspondentes certificados raiz) devidamente instalados: $\text{Cert}_{\text{CA}_1}^{\text{CA}_1} \dots \text{Cert}_{\text{CA}_n}^{\text{CA}_n}$. Estes certificados servem para garantir a confiança sempre que código *Javascript* assinado é enviado para o *browser Web* do utilizador.

Sempre que é enviado código *Javascript* assinado para o *browser Web*, o *proxy* intercepta esse mesmo conteúdo, verificando o mesmo indicado pelo facto de o *Javascript* estar protegido, e desencadeia um conjunto de ações para verificar a confiança nesse mesmo código. Esse código estará devidamente assinado digitalmente e identificado por uma credencial digital emitida por uma autoridade de certificação, conforme aquilo que foi anteriormente descrito.

3.2.5. Proteção da Integridade e Garantia da Confiança

Este é o tipo de utilização mais comum que irá ser utilizada pelos programadores, que permitirá verificar a integridade e a confiança do código que será posteriormente enviado para o *browser* do utilizador. Neste caso, o que vai acontecer é que os scripts de *Javascript* quer locais ($\text{DSig}_{\text{Script}_m}^{\text{SD}_n}$) quer remotos ($\text{DSig}_{\text{RScript}_m}^{\text{SD}_n}$), estão assinados digitalmente e vão ter que ser validadas pelo *proxy* antes de serem efetivamente enviadas para o *browser Web* do utilizador ou, no caso de não poderem ser validadas, serem descartadas pelo *proxy*.

Para efetuar a validação da integridade dos *scripts* de *Javascript* que estão integrados nos ficheiros de HTML que compõem a aplicação *Web*, são usados os certificados digitais das entidades produtoras de código (programadores ou empresas) que contém a correspondente chave pública que permite efetuar a validação.

Este processo passa pelo seguinte:

1. Extração do certificado digital, que está integrado no ficheiro HTML da aplicação *Web*: $\text{Cert}_{\text{SD}_n}^{\text{CA}}$.
2. Validar este certificado, verificando que a autoridade de certificação (CA) que o emitiu faz parte da lista de autoridades de certificação nas quais o *proxy* confia. Se por acaso, o certificado recebido, não tiver sido emitido por uma CA de topo (por exemplo, $\text{Cert}_{\text{SD}_1}^{\text{SDCA}}$) toda a cadeia de certificados terá que ser efetivamente

Execução confiável de código Javascript remoto

validada. Para garantir a segurança no certificado digital, é ainda possível a utilização de *Online Certificate Status Protocol* (OCSP) [42] para auscultar a possibilidade de existirem certificados que, entretanto, tenham sido revogados.

3. Finalmente, depois de estabelecida a confiança na entidade emissora do certificado digital, e, conseqüentemente, no certificado digital recebido, é possível confiar na chave pública que o mesmo contém (K_{SD}^{pub}).
4. Esta chave pública pode ser então utilizada para verificar a assinatura digital do ficheiro HTML.

$$VDSig_{HTML}^{SD_n} \rightarrow K_{SD_n}^{pub}(DSig_{HTML}^{SD_n}), \{n \in \mathbb{Z} | 1 \leq n \leq \infty\}$$

5. Depois de ter sido validada a assinatura digital do ficheiro HTML, é depois necessário, validar as assinaturas digitais de todos os scripts remotos e locais da aplicação Web.

$$VDSig_{Script_m}^{SD_n} \rightarrow K_{SD_n}^{pub}(DSig_{Script_m}^{SD_n}), \{n, m \in \mathbb{Z} | 1 \leq n, m \leq \infty\}$$

$$VDSig_{RScript_m}^{SD_n} \rightarrow K_{SD_n}^{pub}(DSig_{RScript_m}^{SD_n}), \{n, m \in \mathbb{Z} | 1 \leq n, m \leq \infty\}$$

6. Depois de terem sido validadas as diversas assinaturas digitais dos scripts locais e remotos, os mesmos podem depois ser enviados para o browser Web para poderem ser executados.

3.2.6. Proteção da Confidencialidade e da Propriedade Intelectual

Este processo é um processo adicional de segurança que permite proteger a confidencialidade dos próprios scripts de *Javascript*, garantindo igualmente a proteção da propriedade intelectual dos produtores de *software*. Depois de já ter sido executado o processo descrito anteriormente, isto é, de ter sido analisado e verificada a confiança e integridade do próprio recurso de HTML, o *proxy* já está na posse do certificado digital ($Cert_{SD_n}^{CA}$) que contem a chave pública da entidade que envia a aplicação *Web* ($K_{SD_n}^{pub}$).

Depois de possuir esta chave pública, o *proxy* vai enviar um novo pedido para o servidor (através de um pedido HTTP POST), em que após escolher uma chave criptográfica secreta (S_k), a vai cifrar com a chave pública do servidor e a envia para o mesmo ($K_{SD_n}^{pub}(S_k)$).

Execução confiável de código Javascript remoto

Face a isto, o programador vai decifrar esta chave, recorrendo à sua própria chave privada ($K_{SD_n}^{priv} \left(K_{SD_n}^{pub}(S_k) \right) \rightarrow S_k$). Esta chave é usada para cifrar os diversos scripts (embora dependa do modelo que foi descrito anteriormente): $\cup_{i=1}^n S_k(\text{Script}_i)$.

Estes *scripts* protegidos são enviados para o *proxy*, que recorrendo à mesma chave secreta (S_k), vai proceder à decifra dos mesmos antes de os passar ao *browser Web*: $\cup_{i=1}^n S_k(S_k(\text{Script}_i))$.

Fica assim garantida a confidencialidade *end-to-end* dos *scripts* de *Javascript*, e consequentemente a propriedade intelectual dos programadores.

3.3. ScriptProtector

Conforme foi anteriormente referido, o “ScriptProtector” é o sistema que estará em funcionamento do lado do produtor de código *Javascript*, como forma de oferecer as garantias de confiança e de proteção do mesmo, aquando da sua execução por um cliente.

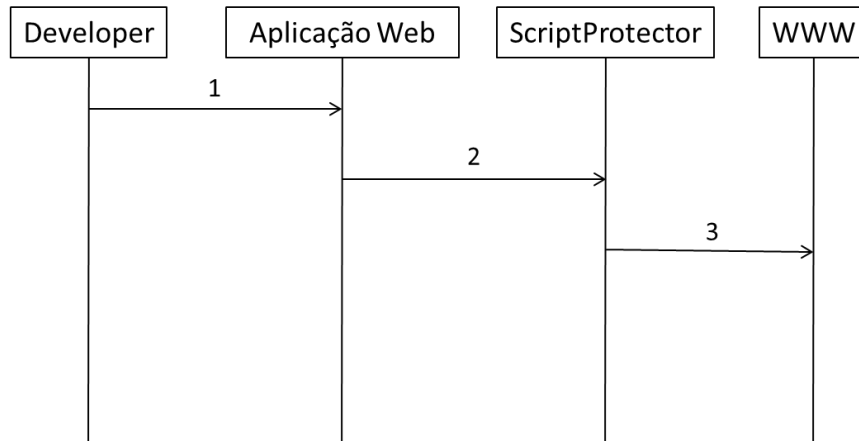


Figura 8 - Diagrama de sequência das interações no ciclo de vida do ScriptProtector

Na figura 8, podemos identificar as interações entre os elementos que compõem o ciclo de vida de utilização do sistema “ScriptProtector”:

1. Desenvolvimento da aplicação *Web* por parte do seu criador (*developer*);
2. Disponibilização, da aplicação *Web*, para o sistema “ScriptProtector”, onde serão executados todos os procedimentos de proteção do código *Javascript*;
3. Publicação da aplicação *Web*, devidamente protegida, através da *WWW*.

A arquitetura técnica do “ScriptProtector” está representada na seguinte imagem (Figura 9), assim como os seus principais componentes. Este sistema recebe a indicação da localização de uma determinada aplicação *Web*, e processa a mesma de forma a encontrar os *scripts* de *Javascript* que necessitam de ser protegidos. Estes *scripts* podem fazer parte da aplicação local ou então estarem remotamente localizados, sendo que o “ScriptProtector” será igualmente responsável por tratar da sua proteção. Uma vez identificados todos os *scripts* que necessitam de ser protegidos, os mesmos são protegidos e devidamente processados para que o cliente os possam identificar e executar em confiança.

Os principais componentes do “ScriptProtector” são os seguintes:

- **HTML Parser:** este é um dos componentes do “ScriptProtector” que é responsável por efetuar uma análise dos ficheiros HTML da aplicação *Web* (caso a aplicação use código dinâmico, como por exemplo PHP [43] ou outra tecnologia similar, poderá ser igualmente utilizado), e que verifica a estrutura do mesmo, verificando a existência de código *Javascript* (quer inline, quer remoto) e que vai enviar a referência do mesmo para o componente “Script Parser”.
- **Script Parser:** este componente é responsável por, após ter recebido a notificação por parte do “HTML Parser” da existência de *scripts* locais ou remotos, de os identificar e depois extrair para posterior proteção. Se alguns destes *scripts* não estiverem disponíveis localmente (*inline*), duas decisões podem ser tomadas. Se a política de proteção a utilizar decidir que todos os *scripts* de *Javascript* devem estar protegidos (locais e remotos), então os mesmos são obtidos/extraídos para posterior proteção, sendo que este componente irá recorrer igualmente ao “Network Comm” para os obter. No caso desta proteção global não ser necessária, apenas os *scripts* de *Javascript* locais são obtidos para posterior proteção.
- **Network Comm:** este componente é responsável pela obtenção dos *scripts* de *Javascript* remotos que estão relacionados com a aplicação *Web*.
- **Script Protection:** este componente é responsável pelo processo de proteção dos *scripts* de *Javascript* na página HTML da aplicação *Web*. Este componente recorre ao “Encryptor” para efetuar o esquema de proteção adequado para garantir que o *script* de *Javascript* irá cumprir com eventuais critérios de confidencialidade e integridade que são necessários para garantir a confiança ao utilizador.
- **Encryptor:** componente que é responsável pelos processos criptográficos envolvidos na proteção do *script* de *Javascript*, recorrendo a criptografia de chave secreta e criptografia de chave pública para garantir a confidencialidade e integridade do mesmo.

- **Script Output:** Finalmente, este componente é responsável pelo *output* das páginas HTML (ou semelhantes) que contém os *scripts* de *Javascript* devidamente protegidos, num formato que pode garantir ao “ScriptProxy” a garantir de confidencialidade e de autenticação dos scripts.

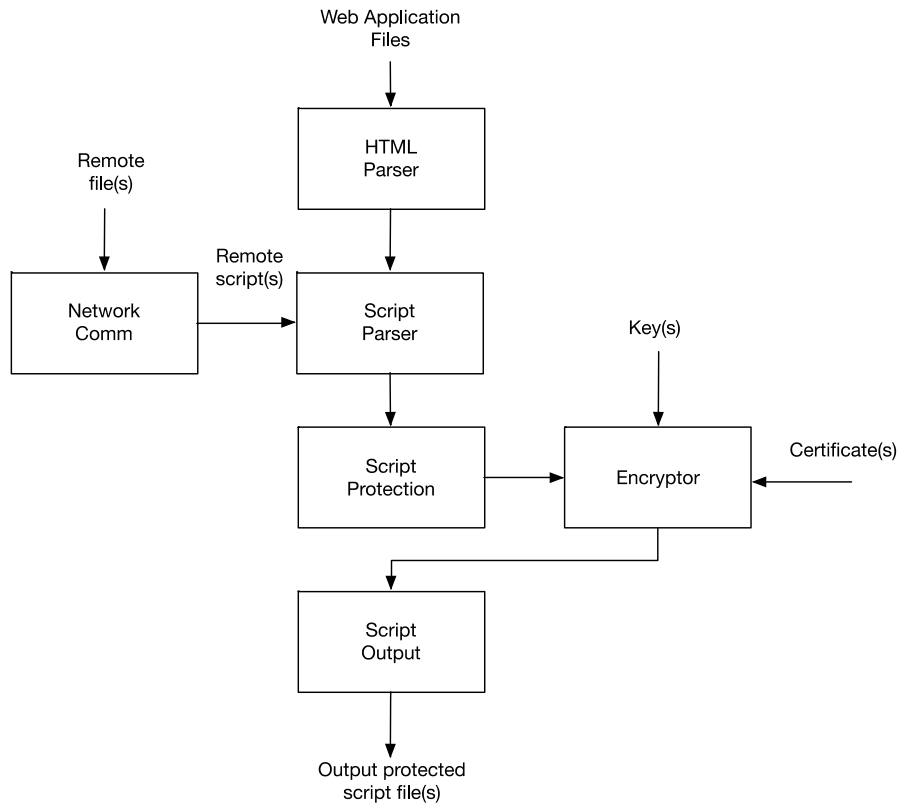


Figura 9 - Arquitetura técnica do ScriptProtector

3.3.1. Implementação do ScripProtector

Como prova de conceito foi desenvolvida uma aplicação *Java* [44], que o produtor da aplicação *Web* deve executar, anteriormente à sua disponibilização, para conseguir assim preparar a mesma com os elementos necessários para o funcionamento correto com o “ScriptProxy”. Em seguida encontra-mos a descrição da implementação da mesma, sendo que o seu código fonte se encontra na secção Anexo A.

Esta aplicação utiliza uma API de manuseamento HTML (*HTML parser*), designada por “*jsoup*” [45], que permite a identificação e manuseamento dos elementos DOM de uma forma mais direta e eficiente.

Ao executar esta aplicação, são solicitados, ao produtor, os diretórios dos ficheiros PEM [46] da sua chave privada, certificado (que contém a chave pública, obtida a partir de

Execução confiável de código Javascript remoto

uma CA, segundo procedimento descrito anteriormente) e do ficheiro *HTML* que pretende preparar/proteger.

Estando na posse destes três elementos, a aplicação vai, para cada elemento *Javascript* encontrado, criar um formulário com o identificador "formSig", que irá conter:

- Um elemento *hidden* (escondido para o utilizador) com a assinatura digital, com o identificador "siggenerated";
- Um elemento *hidden* com a chave pública do produtor, com o identificador "cert";
- O próprio conteúdo *Javascript* com a adição de uma classe específica ("js") no elemento que o contém "<script ...", permitindo assim que a extensão identifique e aceda ao mesmo.

Os identificadores presentes em cada formulário irão ser concatenados com um número inteiro, de valor inicial igual a zero, e que será incrementado por cada elemento *Javascript* existente, ou seja, o primeiro formulário terá o identificador "formSig0", o segundo "formSig1", e assim sucessivamente.

Para que a assinatura de cada elemento *Javascript* seja realizada, primeiramente, é convertida chave privada e para um *array* de *bytes*, utilizando biblioteca `BASE64Decoder` [47] e encriptada recorrendo ao algoritmo criptográfico RSA:

```
// generate private key
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
RSAPrivateCrtKeySpec keySpec = getRSAKeySpec(keyBytes);
return keyFactory.generatePrivate(keySpec);
```

Sendo o elemento "keyBytes" a chave privada no formato de *array* de *bytes* e as classes "KeyFactory" e "RSAPrivateCrtKeySpec", classes *Java* pertencentes ao `java.security`, existentes no *JRE 1.8*, utilizado nesta aplicação.

De seguida é iniciada uma classe *Signature*, presente também no pacote `java.security`, que será a assinatura digital, com formato RSA numa estrutura *SHA1*

Execução confiável de código Javascript remoto

[48] (*hash* de 20 *bytes*/hexadecimal de 40 dígitos), tudo isto utilizando a referência "SHA1withRSA":

```
Signature dsa = Signature.getInstance("SHA1withRSA");
```

Finalmente, para cada conteúdo *Javascript* é gerada uma assinatura digital (*signaux*). Para isso, como podemos verificar no código abaixo, primeiramente faz-se o carregamento da chave privada para a classe da assinatura, em seguida o carregamento do conteúdo *Javascript*, convertido em *bytes*, e por fim a geração da assinatura em si:

```
dsa.initSign(privateKey);  
dsa.update(script.html().trim().getBytes());  
byte[] signaux = dsa.sign();
```

Esta assinatura é devolvida no formato de *array* de *bytes*, que é depois convertido para formato hexadecimal.

Obtemos assim a página HTML com os elementos *Javascript* no interior de formulários, que contêm também o certificado e a assinatura digital, respetivos. Encontrando-se assim, compatível com o funcionamento da extensão no *browser*, é devolvido um ficheiro HTML para o mesmo diretório do ficheiro original, mas com o acrescento "Signed" no nome.

3.4. ScriptProxy

O “ScriptProxy” é o sistema que estará em funcionamento directamente no lado do cliente que irá receber o código da aplicação *Web* para posteriormente ser executado pelo *browser Web*, mas que executa um conjunto de operações que permitem verificar a confidencialidade e a integridade dos scripts que serão posteriormente executados pelo *browser Web*.

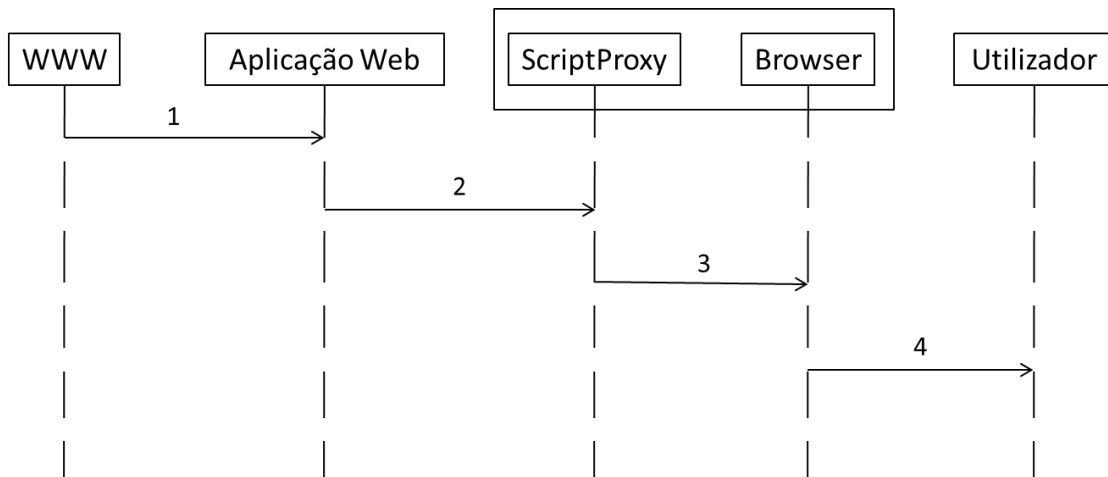


Figura 10 – Diagrama de sequência das interações no ciclo de vida do ScriptProxy

Na figura 10 podemos identificar as interações entre os elementos que compõem o ciclo de vida do processo onde se encontra o “ScriptProxy”:

1. Identificação na rede da aplicação *Web* pretendida (acesso a uma determinada URL);
2. Envio dos componentes da aplicação *Web* para o *client-side*, especificamente para o “ScriptProxy” onde vão ser feitas as validações necessárias;
3. Depois das validações terem sido efectuadas, os componentes da aplicação *Web* são enviados para o *browser Web*;
4. Finalmente o conteúdo é visualizado pelo utilizador, podendo este interagir com o mesmo.

A arquitetura técnica do “ScriptProxy” está representada na seguinte imagem (Figura 11), assim como os seus principais componentes. Este sistema é responsável por interceptar as páginas que são solicitadas pelo *browser Web*, atuando como um intermediário entre o *browser Web* e o servidor da aplicação *Web*, permitindo analisar

os *scripts* que estavam protegidos no código e implementando os mecanismos necessários que permitem verificar a segurança dos mecanismos de confidencialidade e de confiança estabelecidos.

Os principais componentes do “ScriptProxy” são os seguintes:

- **HTML Parser:** este componente do “ScriptProxy” é responsável por analisar o conteúdo dos ficheiros da aplicação Web que estão protegidos, perceber a estrutura dos mesmos e perceber quais os mecanismos de proteção que foram aplicados e perceber a forma como os mesmos podem ser validados.
- **Network Comm:** este é o módulo que será responsável pelo envio e receção de informação na rede de comunicações, e que permite enviar pedidos e receber o conteúdo da aplicação Web que será posteriormente processada. O conteúdo da aplicação recebido por este componente será posteriormente passado para o “HTML Parser” para poder ser processado.
- **Script Parser:** este módulo do “ScriptProxy” é o responsável por analisar os scripts em *Javascript* da aplicação Web que são identificados pelo “HTML Parser”. O “Script Parser” identifica os scripts de Javascript que foram alvo de proteção e encaminha os mesmos para o módulo de “Script Validation” para que possam ser processados e validados.
- **Script Validation:** este módulo vai verificar os mecanismos de validação que foram aplicados pelo distribuidor do script de Javascript e vai tentar validar os mesmos recorrendo ao módulo “Decryptor”. Depois de validar devidamente os scripts e de ter estabelecidos os princípios de confiança e de confidencialidade necessários, através do módulo “Decryptor” passa o resultado ao “Script Output”.
- **Decryptor:** componente que é responsável pelos processos criptográficos envolvidos na validação do script de Javascript, recorrendo a criptografia de chave secreta e criptografia de chave pública para garantir a confidencialidade e integridade do mesmo.

- **Script Output:** este módulo do “ScriptProxy” é responsável por passar os scripts de Javascript, devidamente validados, em conjunto com a restante informação da aplicação Web para o browser Web para que o mesmo as possa executar.

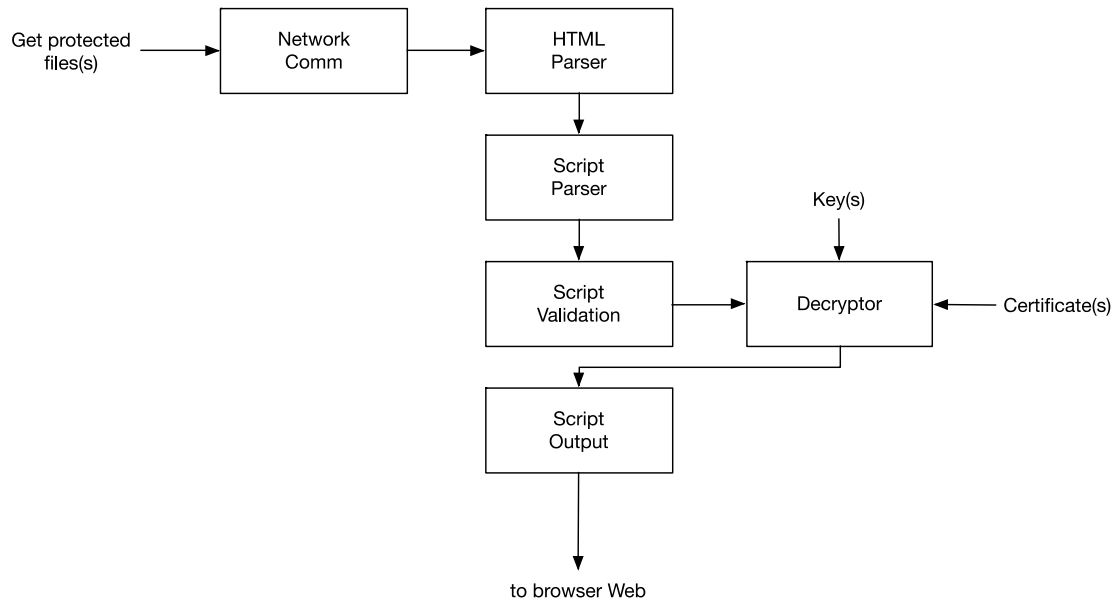


Figura 11 – Arquitetura técnica do ScriptProxy

3.4.1. Implementação do ScriptProxy

No entanto, para a elaboração de uma prova conceito do “ScriptProxy”, por uma questão de integração e de melhorar a experiência de utilização, foi decidido desenvolver o mesmo em formato de extensão para o *browser Chrome*, aproveitando assim o ponto de execução em que este se encontra, e com ele, todos os recursos que este providencia (chamadas e respostas Web). Em seguida encontra-mos a descrição da implementação da mesma, sendo que o seu código fonte se encontra na secção Anexo B.

Uma extensão é um programa de *software* que altera a funcionalidade do *browser*. No caso particular de uma extensão para o *browser Web Google Chrome* (ou *Chromium*) esta é escrita em linguagens como HTML, *Javascript* e CSS. Todos os componentes/ficheiros que compõem a extensão são tornados num só e posteriormente instalados no *browser Web*.

Havendo um menor número de extensões com funções semelhantes, às propostas no sistema a desenvolver, e sendo de maior acessibilidade a implementação no *browser Google Chrome* (método de desenvolvimento de mais fácil aprendizagem/adaptação,

Execução confiável de código Javascript remoto

comparando com o *browser Firefox*), foi decidido desenvolver inicialmente para *browser Google Chrome*, sendo depois possível fazer a adaptação para *browsers* da família *Mozilla*.

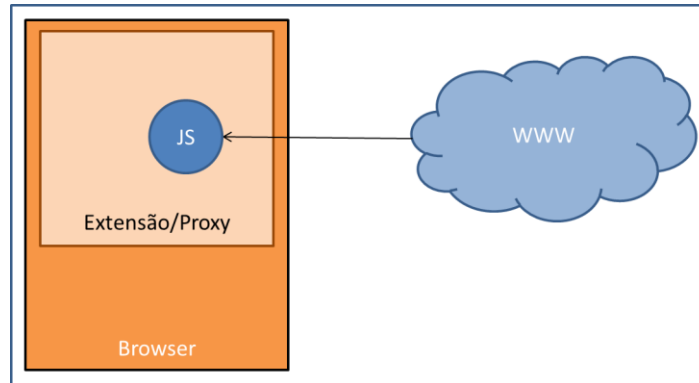


Figura 12 - Esquema simplificado da solução a desenvolver

Na figura 12, podemos observar um esquema simplificado da solução a desenvolver. As respostas *web*, anteriormente ao carregamento no *browser* do utilizador, são interceptadas pela extensão/*proxy* (ScriptProxy) e efectuada a análise ao *Javascript*.

Como foi anteriormente referido, a extensão desenvolvida assume o papel de *proxy* entre o *browser* do utilizador e a *Web*. Esta, terá a capacidade de interceptar o carregamento nas páginas *Web* no *browser*, executar a validação do *Javascript*, e em seguida, com base no resultado dessa validação, prosseguir, ou não, com o carregamento da página *Web*.

O código *Javascript* será validado, comparando a assinatura digital (desencriptada com base na chave pública fornecida) e o conteúdo do próprio *Javascript*, existentes na página. Se o resultado dessa validação for positivo, ou seja, o código *Javascript* é idêntico ao que foi assinado pelo fornecedor da página, prossegue com o carregamento normal da mesma. Em caso negativo (o *Javascript* estar diferenciado/alterado do original), o utilizador será questionado, com uma janela de alerta, se pretende prosseguir com carregamento da página, mesmo sabendo que esta foi modificada desde a sua origem. No caso de não aceitar, o utilizador é retornado para a página anterior à qual se encontra.

Na elaboração de uma extensão *Google Chrome*, o seu primeiro e principal componente é o ficheiro “*manifest.json*”. Este é um ficheiro de *meta* dados em formato JSON

Execução confiável de código Javascript remoto

que contém todas as propriedades da extensão (nome, descrição, versão, entre outros), bem como, as permissões que vai ter e os ficheiros que irá utilizar.

Como queremos que esta extensão seja executada no contexto das páginas *Web*, é utilizado um “content script”, que é um ficheiro *Javascript* que utiliza o *DOM* para ler/modificar os detalhes das páginas que o utilizador acede. Então, para além das propriedades acima identificadas, no “manifest” definimos o “content script” da seguinte forma:

```
"content_scripts": [  
  {  
    "matches": ["http://*/**", "https://*/**"],  
    "run_at": "document_end",  
    "js": [  
      "jquery-1.11.3.min.js",  
      "jsbn.js",  
      "jsbn2.js",  
      "rsa.js",  
      "rsa2.js",  
      "base64.js",  
      "yahoo-min.js",  
      "core.js",  
      "md5.js",  
      "sha1.js",  
      "sha256.js",  
      "ripemd160.js",  
      "x64-core.js",  
      "sha512.js",  
      "rsapem-1.1.js",  
      "rsasign-1.2.js",  
      "asn1hex-1.1.js",  
      "x509-1.1.js",  
      "crypto-1.1.js",  
      "content.js"  
    ]  
  }  
],
```

No campo “matches” definimos um *array* com o tipo de páginas em que o “content script” deve despoletar, neste caso em todas as páginas com protocolo *HTTP* e *HTTPS*. No campo “run_at”, definimos em que altura do carregamento da página, o “content script” deve ser executado, sendo que a propriedade “document_end” se refere ao facto de ser executado apenas quando o *DOM* já tiver sido carregado, mas quando ainda não tiverem sido carregadas imagens ou *frames*. Quanto ao campo “js”, encontramos todos os ficheiros *Javascript* que devem ser injetados nessas páginas, de modo a que consigam ser acedidos e utilizados pelo “content.js”, onde teremos o código de validação do *Javascript* da página acedida. Neste *array*, apenas o “content.js” terá a ação “essencial” da extensão (operações de validação), todos os outros ficheiros *Javascript* são bibliotecas necessárias para a mesma, sendo o “jquery-1.11.3.min.js” usado apenas para simplificar a escrita do código *Javascript* e as restantes bibliotecas utilizadas pela “jsrsasign” [49], sendo esta uma *API Javascript* de criptografia RSA.

O “content script” lê todo o *HTML* da página, que está a ser carregada no *browser*, e para cada elemento de *Javascript* existente (“<script ...”), vai verificar se existe associado, a este, um elemento com a assinatura digital do conteúdo *Javascript* e um segundo elemento com o certificado, que por sua vez contem a chave pública, do produtor da página. Esta associação é feita pelo executável criado como segundo componente desta dissertação, como vamos ver no ponto a seguir.

Obtenção do conteúdo *Javascript*:

```
var sMsg = $('#'+formJs).html().trim();
```

Obtenção da assinatura:

```
var hSig = $('#siggenerated'+i).val();
```

Obtenção do certificado:

```
var cert = $('#cert'+i).val();
```

Sendo o “i” um número inteiro que irá ser incrementado cada elemento *Javascript* existente.

Execução confiável de código Javascript remoto

Em seguida, com base nos dados obtidos, irá ser feita a validação da assinatura digital, ou seja, descriptando a assinatura digital, com base na chave pública fornecida, se o obtido for igual ao conteúdo *Javascript* relacionado, então este não sofreu modificações, desde a sua criação, e é de confiança. Caso os elementos referidos não existam, ou o conteúdo do *Javascript* não seja idêntico ao descriptado, é exibido um *popup* de confirmação se o utilizador deseja prosseguir com o carregamento da página, caso este não aceite, é remetido para a página onde se encontrava anteriormente.

Função de validação do *Javascript*:

```
function doVerify(sMsg, hSig, cert) {  
  
    var x509 = new X509();  
    x509.readCertPEM(cert);  
    var isValid = x509.subjectPublicKeyRSA.verifyString(sMsg,  
hSig);  
  
    return isValid;  
}
```

A função acima demonstrada, tem como parâmetros o conteúdo *Javascript* (*sMsg*), a assinatura digital (*hSig*) e o certificado que contém a chave pública (*cert*).

É iniciada uma variável “x509” com a classe *X509()* [50], que, por sua vez, fornece o formato *standard* usado nas *PKI's* (infraestruturas de chave-pública), e onde é carregada a mesma, na função “*x509.readCertPEM(cert)*”.

Em seguida é realizada a decifra e validação da assinatura e do conteúdo *Javascript* na função “*x509.subjectPublicKeyRSA.verifyString(sMsg, hSig)*”, encontrando-se esta implementada na *API* “*jsrsasign*”, bem como a função “*readCertPEM*”.

Capítulo 4

Demonstração de
funcionamento

Ao longo deste capítulo será apresentada a validação do sistema desenvolvido, através da demonstração do funcionamento do mesmo, isto é, da aplicação *Java*, “ScriptProtector”, e da extensão para o *browser Chrome*, “ScriptProxy”, como provas de conceito. É apresentado ainda um estudo do caso, em que, em função dos resultados da validação do sistema desenvolvido, permite definir e apresentar os benefícios e falhas detetadas no sistema.

4.1. Utilização do “ScriptProtector”

Para validar esta componente do sistema, foi utilizado um par de chaves assimétricas, que se encontravam disponíveis no código de demonstração fornecido pela API “jsrsasign”, apenas para fins de teste/validação do sistema, e uma página HTML básica, apenas com um título, um elemento *Javascript* e um cabeçalho no corpo da página, podemos observar esta página na figura 13:

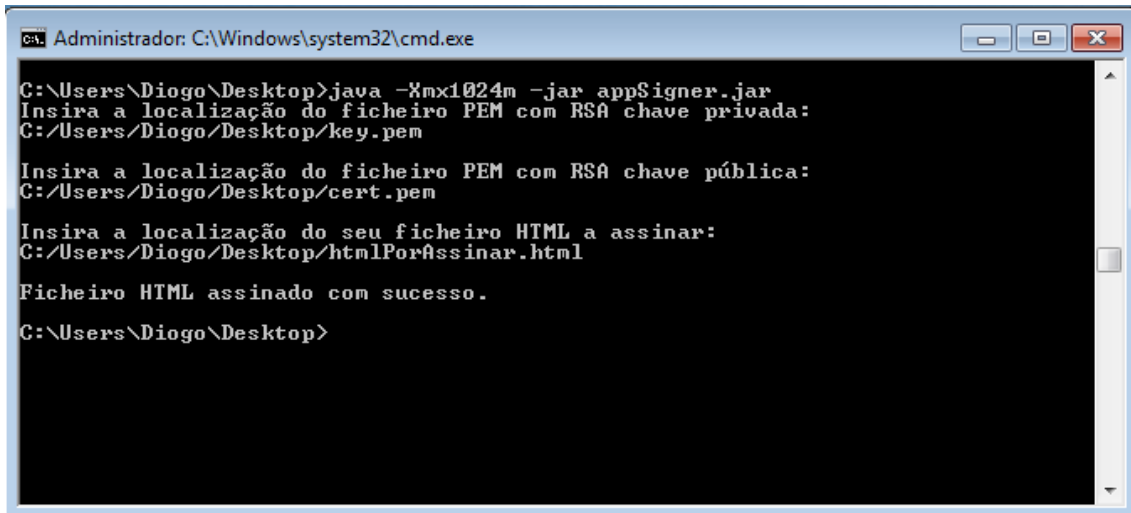
```
<html>
  <head>
    <title>Page Sample for RSA verify in JavaScript</title>

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

Figura 13 – Página HTML para assinar

Para esta prova de conceito, o executável de preparação da página HTML encontra-se em formato “.jar” [51]. Para ser possível executar o mesmo (por exemplo, utilizando o sistema operativo Windows 7 [52]), o utilizador acede à linha de comandos do sistema operativo e, encontrando-se na diretoria onde o executável se encontra, executa o comando “java -Xmx1024m -jar appSigner.jar”, sendo o elemento “appSigner.jar” o executável *Java* em si. De seguida indica a diretoria das chaves assimétricas e da página a assinar, concluindo o ciclo de vida do executável. Podemos observar as ações descritas demonstradas na figura 14:

Execução confiável de código Javascript remoto



```
C:\Windows\system32\cmd.exe

C:\Users\Diogo\Desktop>java -Xmx1024m -jar appSigner.jar
Insira a localização do ficheiro PEM com RSA chave privada:
C:/Users/Diogo/Desktop/key.pem

Insira a localização do ficheiro PEM com RSA chave pública:
C:/Users/Diogo/Desktop/cert.pem

Insira a localização do seu ficheiro HTML a assinar:
C:/Users/Diogo/Desktop/htmlPorAssinar.html

Ficheiro HTML assinado com sucesso.

C:\Users\Diogo\Desktop>
```

Figura 14 – Executável de preparação/assinatura da página HTML

Assim, é devolvida, na mesma diretoria, a página HTML original devidamente assinada e preparada para funcionar em sincronia com a extensão desenvolvida, demonstrada na figura 15.

```
<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-----MIIBvTCCASYCCQD55fNzo0WF7
    <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>
```

Figura 15 – Página HTML devidamente assinada e preparada

4.2. Utilização do ScriptProxy

Para ativar a extensão *Chrome* desenvolvida, “ScriptProxy”, acedemos às definições do *browser Chrome*, e através da secção de “Extensões”, procedemos ao carregamento da mesma, selecionando a opção “Carregar extensão descomprimida...”, e selecionamos a *checkbox* “Ativada”, para ativar a mesma. Podemos observar o exemplo desta ação na figura 16.

Execução confiável de código Javascript remoto

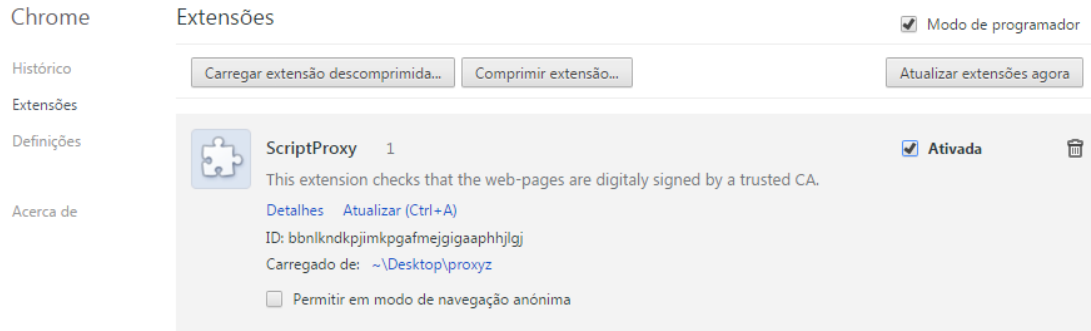


Figura 16 – Ativação da extensão desenvolvida

A partir deste momento a extensão encontra-se ativa, no *browser*, e à “escuta” de respostas que cheguem do acesso a páginas *web*, isto é comprovado com o aparecimento do *icon* na barra de ferramentas do *browser*, demonstrado na figura 17.



Figura 17 – Extensão desenvolvida ativa

Assim, encontrando-se a extensão ativa, quando o utilizador acede a uma página *web* que não esteja com o formato desejado (não se encontra com os elementos *Javascript* existentes na página devidamente assinados), este é alertado, como exemplificado na figura 18:

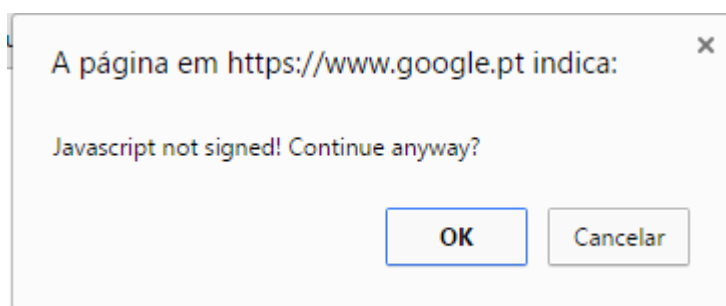


Figura 18 – Alerta ao utilizador quando o Javascript não se encontra assinado

4.3. Estudo do caso

Ao navegarmos com a extensão desenvolvida ativa no *browser*, em qualquer aplicação *Web* acedida, é despoletado o aviso que o *Javascript* presente na aplicação não se encontra assinado, pois nenhuma destas aplicações se encontram devidamente preparadas com os elementos necessários à validação.

Para validação do sistema, é necessário recorrer a uma aplicação *Web* com a estrutura e assinatura dos seus elementos *Javascript*, de acordo com o funcionamento do sistema desenvolvido, e em seguida, simular um ataque de modificação maliciosa do *Javascript* e validar que o sistema deteta essa modificação e garante a segurança do utilizador.

De modo a atingir esse objetivo, foi utilizada uma aplicação que, na sua essência, contém um botão com a função de despoletar uma *popup* com a informação “Original Code!”. Depois de devidamente formatada e assinada pelo “ScriptProtector”, figura 19, foi colocada num serviço de *hosting* e validado que ao aceder, à mesma, o utilizador não é alertado pelo “ScriptProxy”, ou seja, o sistema valida a integridade da assinatura e do conteúdo *Javascript*, presentes na aplicação.

```

<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-----MIIBvTCCASYCCQD55fNzc0WF7TANBgkqhkiG9w0BAQ
<input type="hidden" name="siggenerated0" id="siggenerated0" value="496dc5fadc2c51d83c2ddc4ad2570994ac00eac3059bbaca052:
<script language="JavaScript" type="text/javascript" class="js">function buttonFunc(){alert('Original Code!');}</script>
</form>
</head>
<body>
<h1>Page Sample for RSA verify in JavaScript</h1>
<button type="button" onclick="buttonFunc();">Click Me!</button>
</body>
</html>
    
```

Figura 19 – Página HTML assinada pelo “ScriptProtector”

Aquando do clique no botão existente na aplicação, é despoletada a *popup* com a informação “Original Code!”, figura 20:

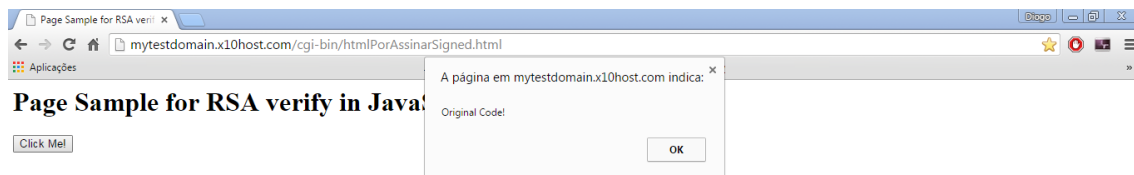


Figura 20 – Clique no botão existente na página HTML

Execução confiável de código Javascript remoto

Em seguida, premindo o botão “*Forward*” na *Burp Suite*, é enviada a resposta modificada para o *browser Web*, completando o ataque *MITM*.

Ao contrário do que aconteceu, no acesso à aplicação anterior à sua modificação, o “*ScriptProxy*” alerta o utilizador que o *Javascript* não se encontra assinado e se deseja prosseguir com o acesso à página (Figura 22). Ou seja, o “*ScriptProtector*”, verificou que o conteúdo *Javascript* não corresponde à assinatura presente na aplicação, alertando o utilizador, garantindo a sua segurança, e demonstrando assim o correcto funcionamento do sistema.

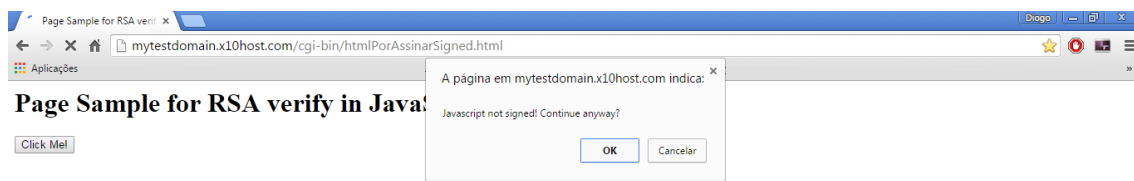


Figura 22 – Alerta de conteúdo Javascript não válido

Assim verificamos que o executável *Java*, “*ScriptProtector*”, prepara e assina devidamente a página *web* pretendida, com base nas chaves assimétricas do cliente, e a extensão desenvolvida, “*ScriptProxy*”, encontra-se em correto funcionamento com o *browser Chrome*, bloqueando as respostas de aplicações *Web* que não contêm os elementos *Javascript* devidamente assinados, ou prosseguindo com a execução da mesma caso esta se encontre assinada e válida. Com base neste resultado, é então garantida a integridade do código desde a sua origem à sua execução no *browser Web*, e confirmando assim que o objetivo proposto nesta dissertação foi atingido com sucesso.

Capítulo 5

Conclusões e Trabalho Futuro

Neste capítulo são apresentadas as principais conclusões obtidas do resultado do trabalho realizado, bem como sugestões para trabalho futuro para eventuais melhorias e funcionalidades a acrescentar no sistema.

5.1. Principais conclusões

É importante realçar que um dos principais objetivos desta dissertação foi alcançado. Foi possível desenvolver um sistema que recorre a mecanismos de assinatura digital, baseados em chaves assimétricas, que garante a origem e a integridade do código *Javascript* das páginas *web* acedidas, desde a sua origem à execução no *browser* do utilizador final. Por outro lado, e se assim o desejarem, os produtores de código podem igualmente usar o sistema desenvolvido para proteger a confidencialidade do próprio código *Javascript* e da correspondente propriedade intelectual.

Esta dissertação tinha como objetivos definidos:

- Identificar os principais aspetos relacionados com questões de problemas de segurança na *Web*, em particular no que diz respeito à execução de código *Javascript* remoto. Este ponto foi estudado na revisão da literatura, onde foram apresentados vários tipos de ataques ao *Javascript*, existentes e mais influentes nos dias que correm.
- Avaliar mecanismos internos e externos aos *browsers Web* que permitam garantir a confiança e integridade na fonte do *Javascript*. Este ponto foi igualmente atingido na revisão da literatura, onde foram apresentados mecanismos de segurança internos e externos aos *browsers Web*, que não só tentam garantir a integridade do código *Javascript* desde a sua fonte, como visam defender o utilizador e fornecedor, do código, de vulnerabilidades existentes no mesmo.
- Desenvolver o sistema que permita garantir a confiança, segurança e integridade do código *Javascript*, desde o momento que é produzido até ao momento em que é executado, recorrendo a mecanismos de criptografia de chave pública. Como referido e apresentado ao longo desta dissertação, foi desenvolvido um sistema, composto por um elemento do tipo *proxy* e uma aplicação *Java*, que recorrem a um mecanismo de chaves assimétricas e que garantem a integridade do conteúdo *Javascript*, desde a sua criação ao carregamento nos *browsers Web*.

- Validar e comparar com os sistemas existentes, que o sistema desenvolvido, realmente pode contribuir para a possível resolução dos problemas de segurança referidos anteriormente. Como podemos validar no ponto Estudo do caso, o sistema desenvolvido, em comparação com os mecanismos de defesa internos e externos apresentados na revisão da literatura, é um método de segurança inovador e eficaz, na garantia de integridade do código *Javascript*.

Consolida-se assim, que o uso de sistemas de chaves assimétricas, para proteção do *Javascript*, é possível, fiável e sem dúvida uma área em que se deve apostar e investigar, devido à falta de oferta de sistemas que recorrem a este método, sendo este de grande potencial. Por outro lado, é importante realçar que o sistema proposto proporciona um nível de confiança adicional, que permite que os *browsers Web* e os utilizadores possam executar e utilizar código remoto confiando na integridade e na origem do mesmo.

Adicionalmente este trabalho proporcionou ainda a possibilidade dos produtores de *software* terem a possibilidade de proteger a sua própria propriedade intelectual, garantindo assim, a confidencialidade dos seus código, quando em trânsito, mesmo quando um canal seguro não pode ser estabelecido entre o *browser Web* e o servidor.

5.2. Sugestões para trabalho futuro

Para além do trabalho realizado e do grande objetivo desta dissertação ter sido alcançado, existe ainda trabalho futuro possível para a melhoria do sistema, que trará novas funcionalidades e maior rigor ao sistema.

Focando-nos na abrangência do sistema desenvolvido, o cenário ideal para a sua execução, como referido no desenho da solução, invés de este se encontrar como uma extensão para o *browser Chrome*, seria, encontrar-se no formato de um programa tipo *proxy* de rede. Ou seja, seria um programa em execução no sistema operativo do utilizador (como proposto inicialmente) que teria como função, sempre que interjetasse a resposta de uma chamada à *Web*, executar a validação presente na extensão desenvolvida. Assim, seria possível processar o *Javascript*, antes de este ser transportado para o *browser*. Este formato teria algumas vantagens, nomeadamente a possibilidade de independência do *browser web*, tornando-o não dependente de uma determinada extensão ou *browser*, assim como permitiria ter um pré-processamento do

Javascript antes de ser executado no *browser*, garantindo maior segurança na altura do carregamento da aplicação.

Focando-nos na funcionalidade do sistema, seria uma mais valia a possibilidade do fornecedor do código *Javascript*, poder definir um conjunto de regras (incluídas na página *web*, por exemplo, em formato *XML*) que permitiriam impor condições sobre a forma de como este pode ser executado do lado do cliente, ou seja, o fornecedor poderia definir um conjunto de regras para cada elemento *Javascript* incluídas na página. Para o elemento *Javascript* “A” seriam dadas condições de execução “X” e para o elemento “B” seriam dadas condições de execução “Y”, limitando assim a execução, e aumentando a segurança, de cada elemento *Javascript*, complementando a funcionalidade do sistema desenvolvido.

Por último, focando-nos na segurança do sistema desenvolvido, perante a ameaça do atacante conhecer o funcionamento do mesmo, o atacante, pode intersetar a resposta do servidor *Web*, como anteriormente exemplificado (com um ataque *MITM* ou outro apresentado no estado da arte), e alterar não só o *Javascript* com o código malicioso pretendido, como também sobrepor a assinatura e o certificado existentes na página, com a sua própria assinatura e certificado (assumindo que o atacante esteja na posse de uma chave assimétrica). A extensão, ao receber a página com os elementos necessários, ao decifrar a assinatura, valida que o conteúdo *Javascript* corresponde à mesma, vai tomar este código como válido/original, e proceder à execução do mesmo.

De modo a ultrapassar esta ameaça, o sistema desenvolvido necessita de aferir a validade do certificado digital (chave pública), utilizado na assinatura digital do *Javascript*. Para tal, o sistema deverá usar o Online Certificate Status Protocol (OSCP) anteriormente à validação do *Javascript*. Para tal, a extensão deverá fazer um pedido de validação do certificado digital a um servidor OCSP, enviando, a este, o certificado. Em seguida, este valida que o certificado não está marcado como válido, que pertence à entidade/página a que o utilizador está a aceder e que se encontra válido, por fim, responde com o resultado desta validação à extensão. Decidindo depois, esta, em função da resposta obtida, que ação deve tomar (se segue com a decifra da assinatura digital, caso o certificado seja válido, ou se bloqueia logo nesta fase o acesso à página, alertando o utilizador).

Execução confiável de código Javascript remoto

Para que estas melhorias sejam realizadas, e estejam ao alcance do público interessado, o código fonte desta dissertação encontra-se publicado no repositório com o URL <https://github.com/ogoid90/JsPkiSafe>.

Referências

- [1] Rosenfeld, L., & Morville, P. (2002). Information architecture for the world wide web. " O'Reilly Media, Inc."
- [2] A Short History of JavaScript. Acedido a 17 de Janeiro de 2015, https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript
- [3] Dhamija, R., Tygar, J. D., & Hearst, M. (2006, April). Why phishing works. In Proceedings of the SIGCHI conference on Human Factors in computing systems (pp. 581-590). ACM.
- [4] Hevner, A., & Chatterjee, S. (2010). *Design science research in information systems* (pp. 9-22). Springer US.
- [5] Kuechler, B., & Vaishnavi, V. (2004). Design Science Research in Information Systems.
- [6] Rosen, R., & Shklar, L. (2003). Web Application Architecture: Principles, Protocols and Practices.
- [7] HTML 4.0 Specification. Acedido a 1 de Outubro de 2015, <http://www.w3.org/TR/REC-html40/>
- [8] Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2), 115-150.
- [9] Haverbeke, M. (2014). *Eloquent JavaScript: A Modern Introduction to Programming*. No Starch Press.
- [10] Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (Vol. 2). Englewood Cliffs: prentice-Hall.
- [11] Lie, H. W., & Bos, B. (2005). *Cascading style sheets: Designing for the web*. Addison-Wesley Professional.
- [12] Nicol, G., Wood, L., Champion, M., & Byrne, S. (2001). Document object model (DOM) level 3 core specification. *W3C Working Draft, 13*, 1-146.
- [13] Teodoro, N., & Serrão, C. (2010). Web applications security assessment in the Portuguese world wide web panorama. In *Web Application Security* (pp. 63-73). Springer Berlin Heidelberg.
- [14] Vogt, P., Nentwich, F., Jovanovic, N., Kirida, E., Kruegel, C., & Vigna, G. (2007, February). Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*.
- [15] Lawton, G. (2007). Web 2.0 creates security challenges. *Computer*, (10), 13-16.

- [16] Burns, J. (2005). Cross Site Request Forgery. *An introduction to a common web application weakness, Information Security Partners.*
- [17] Mao, Z., Li, N., & Molloy, I. (2009). Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security* (pp. 238-255). Springer Berlin Heidelberg.
- [18] Barua, A., Zulkernine, M., & Weldemariam, K. (2013, July). Protecting web browser extensions from javascript injection attacks. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on* (pp. 188-197). IEEE.
- [19] Livshits, B., & Erlingsson, Ú. (2007, June). Using web application construction frameworks to protect against code injection attacks. In *Proceedings of the 2007 workshop on Programming languages and analysis for security* (pp. 95-104). ACM.
- [20] Eriksson, M., & Center, W. G. (2003, October). An example of a man-in-the-middle attack against server authenticated SSL-sessions. In *international conference on applied cryptography and network security.*
- [21] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext transfer protocol--HTTP/1.1* (No. RFC 2616).
- [22] Dougan, T., & Curran, K. (2012). Man in the browser attacks. *International Journal of Ambient Computing and Intelligence (IJACI)*, 4(1), 29-39.
- [23] Barth, A., Jackson, C., Reis, C., & TGC Team. (2008). The security architecture of the Chromium browser. 2010-11-18][2011-7-12]. <http://seclah.stanford.edu/websec/chromium>.
- [24] Dewald, A., Holz, T., & Freiling, F. C. (2010, March). ADSandbox: Sandboxing JavaScript to fight malicious websites. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (pp. 1859-1864). ACM.
- [25] Rydstedt, G., Bursztein, E., Boneh, D., & Jackson, C. (2010). Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web*, 2, 1-13.
- [26] Ruderman, J. (2009). Same origin policy for JavaScript. *Online at [https://developer.mozilla.org/En/Same origin policy for JavaScript](https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript).*
- [27] Barth, A. (2011). The web origin concept. Acedido a 17 de Janeiro de 2015, <http://tools.ietf.org/html/rfc6454>
- [28] Xu, W., Zhang, F., & Zhu, S. (2012, October). The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on* (pp. 9-16). IEEE.
- [29] Wressnegger, C., Boldewin, F., & Rieck, K. (2013). Deobfuscating Embedded Malware using Probable-Plaintext Attacks. In *Research in Attacks, Intrusions, and Defenses* (pp. 164-183). Springer Berlin Heidelberg.

- [30] NoScript. Acedido a 16 de Junho de 2015, <https://noscript.net/>
- [31] Whitelist. Acedido a 1 de outubro de 2015, <https://en.wikipedia.org/wiki/Whitelist>
- [32] Untrusted blacklist. Acedido a 1 de outubro de 2015, <https://noscript.net/features#blacklist>
- [33] Anti-XSS protection. Acedido a 1 de outubro de 2015, <https://noscript.net/features#xss>
- [34] NoScript. Acedido a 17 de janeiro de 2015, <https://noscript.net/>
- [35] Options. Acedido a 1 de outubro de 2015, <https://noscript.net/features#options>
- [36] JScrambler. Acedido a 21 de maio de 2015, <https://jscrambler.com/pt/>
- [37] Neamtiu, I., Foster, J. S., & Hicks, M. (2005). Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4), 1-5.
- [38] A spotlight on JSA, the amazing hybrid security analysis for JavaScript. Acedido a 6 de Julho 2015. <http://blog.watchfire.com/wfblog/2012/06/by-now-you-have-probably-heard-about-jsa-introduced-inappscan-standard80-in-oct-10-jsa-is-a-component-that-does-static-ana.html>
- [39] Jensen, S. H., Møller, A., & Thiemann, P. (2009). Type analysis for JavaScript. In *Static Analysis* (pp. 238-255). Springer Berlin Heidelberg.
- [40] Jones, J. (2008). The RSA algorithm. *ACM Communications in Computer Algebra*, 42(1-2), 74-74.
- [41] Housley, R., & Polk, T. (2001). *Planning for PKI: best practices guide for deploying public key infrastructure*. John Wiley & Sons, Inc..
- [42] Myers, M., Ankney, R., Malpani, A., Galperin, S., & Adams, C. (1999). Online certificate status protocol-OCSP.
- [43] Serrão, C., & Marques, J. (2009). Programação com PHP 5.3. *FCA—Editora de Informática*.
- [44] Joy, B., Steele, G., Gosling, J., & Bracha, G. (2000). Java (TM) Language Specification. *Addison-Wesley, June*.
- [45] jsoup: Java HTML Parser. Acedido a 21 de Maio de 2015. <http://jsoup.org/>
- [46] Kaliski Jr, B. S. (1993). An overview of the PKCS standards. In *RSA Data Security, Inc*.
- [47] Class BASE64Decoder. Acedido a 22 de Maio de 2015. <https://tools.ietf.org/html/rfc1421>
- [48] Rijmen, V., & Oswald, E. (2005). Update on SHA-1. In *Topics in Cryptology—CT-RSA 2005* (pp. 58-71). Springer Berlin Heidelberg.
- [49] Jsrsasn. Acedido a 21 de Maio de 2015. <http://kjur.github.io/jsrsasn/>

[50] Gerck, E. (2000). Overview of Certification Systems: X. 509, PKIX, CA, PGP & SKIP. *The Bell*, 1(3), 8.

[51] Using JAR Files: The Basics. Acedido a 10 de Agosto de 2015.
<https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html>

[52] Boyce, J. (2011). *Windows 7 Bible* (Vol. 610). John Wiley & Sons.

[53] Burp Suite. Acedido a 5 de Setembro de 2015. <https://portswigger.net/burp/>

Anexo A

Código Java do ScriptProtector

Classe TranslatorV1Main

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.math.BigInteger;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.Signature;
import java.security.spec.RSAPrivateCrtKeySpec;
import java.util.Scanner;
import javax.xml.bind.DatatypeConverter;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;
import Decoder.BASE64Decoder;

public class TranslatorV1Main {

    public static final String P1_END_MARKER = "-----END RSA
PRIVATE KEY";
    final protected static char[] hexArray =
"0123456789ABCDEF".toCharArray();

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        String rsaKey;
        String cert;
        String html;

        while(true) {

            System.out.println("Insira a localização do
ficheiro PEM com RSA chave privada:");
            rsaKey = sc.nextLine();
            System.out.println("");

            System.out.println("Insira a localização do
ficheiro PEM com RSA chave pública:");
            cert = sc.nextLine();
            System.out.println("");

            System.out.println("Insira a localização do seu
ficheiro HTML a assinar:");
            html = sc.nextLine();
            System.out.println("");

            try {
                signPage(rsaKey, html, cert);
                System.out.println("Ficheiro HTML assinado
com sucesso.");
            } catch (IOException e) {
```

```

        System.out.println("Erro ao obter o
ficheiro HTML.");
    } catch (Exception e) {
        System.out.println("Erro ao assinar o
ficheiro HTML.");
    }

    sc.close();
    System.exit(0);
}

private static void signPage(String rsaKey, String html,
String cert) throws Exception {
    File pgehtml = new File(html);
    Document doc = Jsoup.parse(pgehtml, "UTF-8", "");

    Scanner sc = new Scanner(new File(cert));
    String certficte = "";
    while(sc.hasNextLine()){
        certficte = sc.nextLine();
    }
    sc.close();

    Elements scripts = doc.getElementsByTag("script");

    if(null != scripts && scripts.size() > 0){

        PrivateKey privateKey = readPrivateKey(new
File(rsaKey));
        Signature dsa =
Signature.getInstance("SHA1withRSA");

        for (int i = 0; i < scripts.size(); i++) {

            Element script = scripts.get(i);
            script.attr("class", "js");

            dsa.initSign(privateKey);
            dsa.update(script.html().trim().getBytes());
            byte[] signaux = dsa.sign();

            Element parent = script.parent();
            parent.appendChild("form").attr("name",
"formSig"+i).attr("id", "formSig"+i);

            Element form =
parent.getElementById("formSig"+i);

            form.prependElement("input").attr("type","hidden").attr("na
me","siggenerated"+i).attr("id","siggenerated"+i).attr("value",
toHexString(signaux).toLowerCase());

            form.prependElement("input").attr("type","hidden").attr("na
me","cert"+i).attr("id","cert"+i).attr("value", certficte);
            form.append(script.outerHtml());

```

```

        script.remove();
    }

    String html2 = html.replace(".html", "");
    html2 = html2+"Signed.html";

    PrintWriter writer = new PrintWriter(html2,
"UTF-8");
    writer.println(doc);
    writer.close();
    }
}

public static String toHexString(byte[] array) {
    return DatatypeConverter.printHexBinary(array);
}

public static PrivateKey readPrivateKey(File keyFile)
throws Exception {
    // read key bytes
    FileInputStream in = new FileInputStream(keyFile);
    byte[] keyBytes = new byte[in.available()];
    in.read(keyBytes);
    in.close();

    String privateKey = new String(keyBytes, "UTF-8");
    privateKey = privateKey.replace("-----BEGIN RSA
PRIVATE KEY-----", "");
    privateKey = privateKey.replace("-----END RSA PRIVATE
KEY-----", "");

    BASE64Decoder decoder = new BASE64Decoder();
    keyBytes = decoder.decodeBuffer(privateKey);

    // generate private key
    KeyFactory keyFactory =
KeyFactory.getInstance("RSA");
    RSAPrivateCrtKeySpec keySpec =
getRSAKeySpec(keyBytes);
    return keyFactory.generatePrivate(keySpec);
}

private static RSAPrivateCrtKeySpec getRSAKeySpec(byte[]
keyBytes) throws IOException {

    DerParser parser = new DerParser(keyBytes);

    Asn1Object sequence = parser.read();
    if (sequence.getType() != DerParser.SEQUENCE)
        throw new IOException("Invalid DER: not a sequence");

    parser = sequence.getParser();

    parser.read();
}

```

```

        BigInteger modulus = parser.read().getInteger();
        BigInteger publicExp = parser.read().getInteger();
        BigInteger privateExp = parser.read().getInteger();
        BigInteger prime1 = parser.read().getInteger();
        BigInteger prime2 = parser.read().getInteger();
        BigInteger exp1 = parser.read().getInteger();
        BigInteger exp2 = parser.read().getInteger();
        BigInteger crtCoef = parser.read().getInteger();

        RSAPrivateCrtKeySpec keySpec = new RSAPrivateCrtKeySpec(
            modulus, publicExp, privateExp, prime1, prime2,
            exp1, exp2, crtCoef);

        return keySpec;
    }
}

```

Class DerParser

(<https://github.com/aws/aws-sdk-java/blob/master/aws-java-sdk-cloudfront/src/main/java/com/amazonaws/auth/DerParser.java>)

```

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.math.BigInteger;

public class DerParser {

    // Classes
    public final static int UNIVERSAL = 0x00;
    public final static int APPLICATION = 0x40;
    public final static int CONTEXT = 0x80;
    public final static int PRIVATE = 0xC0;

    // Constructed Flag
    public final static int CONSTRUCTED = 0x20;

    // Tag and data types
    public final static int ANY = 0x00;
    public final static int BOOLEAN = 0x01;
    public final static int INTEGER = 0x02;
    public final static int BIT_STRING = 0x03;
    public final static int OCTET_STRING = 0x04;
    public final static int NULL = 0x05;
    public final static int OBJECT_IDENTIFIER = 0x06;
    public final static int REAL = 0x09;
    public final static int ENUMERATED = 0x0a;
    public final static int RELATIVE_OID = 0x0d;

    public final static int SEQUENCE = 0x10;
    public final static int SET = 0x11;

    public final static int NUMERIC_STRING = 0x12;
    public final static int PRINTABLE_STRING = 0x13;
}

```



```

public final static int T61_STRING = 0x14;
public final static int VIDEOTEX_STRING = 0x15;
public final static int IA5_STRING = 0x16;
public final static int GRAPHIC_STRING = 0x19;
public final static int ISO646_STRING = 0x1A;
public final static int GENERAL_STRING = 0x1B;

public final static int UTF8_STRING = 0x0C;
public final static int UNIVERSAL_STRING = 0x1C;
public final static int BMP_STRING = 0x1E;

public final static int UTC_TIME = 0x17;
public final static int GENERALIZED_TIME = 0x18;

protected InputStream in;

/**
 * Create a new DER decoder from an input stream.
 *
 * @param in
 *         The DER encoded stream
 */
public DerParser(InputStream in) throws IOException {
    this.in = in;
}

/**
 * Create a new DER decoder from a byte array.
 *
 * @param The
 *         encoded bytes
 * @throws IOException
 */
public DerParser(byte[] bytes) throws IOException {
    this(new ByteArrayInputStream(bytes));
}

/**
 * Read next object. If it's constructed, the value holds
 * encoded content and it should be parsed by a new
 * parser from <code>Asn1Object.getParser</code>.
 *
 * @return A object
 * @throws IOException
 */
public Asn1Object read() throws IOException {
    int tag = in.read();

    if (tag == -1)
        throw new IOException("Invalid DER: stream too
short, missing tag"); //$NON-NLS-1$

    int length = getLength();

    byte[] value = new byte[length];
    int n = in.read(value);

```

```

        if (n < length)
            throw new IOException("Invalid DER: stream too
short, missing value"); //$NON-NLS-1$

        Asn1Object o = new Asn1Object(tag, length, value);

        return o;
    }

    /**
     * Decode the length of the field. Can only support length
     * encoding up to 4 octets.
     *
     * <p/>In BER/DER encoding, length can be encoded in 2
forms,
     * <ul>
     * <li>Short form. One octet. Bit 8 has value "0" and bits
7-1
     * give the length.
     * <li>Long form. Two to 127 octets (only 4 is supported
here).
     * Bit 8 of first octet has value "1" and bits 7-1 give the
     * number of additional length octets. Second and following
     * octets give the length, base 256, most significant digit
first.
     * </ul>
     * @return The length as integer
     * @throws IOException
     */
    private int getLength() throws IOException {

        int i = in.read();
        if (i == -1)
            throw new IOException("Invalid DER: length
missing"); //$NON-NLS-1$

        // A single byte short length
        if ((i & ~0x7F) == 0)
            return i;

        int num = i & 0x7F;

        // We can't handle length longer than 4 bytes
        if (i >= 0xFF || num > 4)
            throw new IOException("Invalid DER: length field
too big (" //$NON-NLS-1$
                + i + ")"); //$NON-NLS-1$

        byte[] bytes = new byte[num];
        int n = in.read(bytes);
        if (n < num)
            throw new IOException("Invalid DER: length too
short"); //$NON-NLS-1$

        return new BigInteger(1, bytes).intValue();
    }
}

```

```

}
class Asn1Object {

    protected final int type;
    protected final int length;
    protected final byte[] value;
    protected final int tag;

    /**
     * Construct a ASN.1 TLV. The TLV could be either a
     * constructed or primitive entity.
     *
     * <p/>The first byte in DER encoding is made of
following fields,
     * <pre>
     * -----
     *|Bit 8|Bit 7|Bit 6|Bit 5|Bit 4|Bit 3|Bit 2|Bit 1|
     *-----
     *|  Class   | CF   |      +      Type           |
     *-----
     * </pre>
     * <ul>
     * <li>Class: Universal, Application, Context or
Private
     * <li>CF: Constructed flag. If 1, the field is
constructed.
     * <li>Type: This is actually called tag in ASN.1. It
     * indicates data type (Integer, String) or a
construct
     * (sequence, choice, set).
     * </ul>
     *
     * @param tag Tag or Identifier
     * @param length Length of the field
     * @param value Encoded octet string for the field.
     */
    public Asn1Object(int tag, int length, byte[] value)
    {
        this.tag = tag;
        this.type = tag & 0x1F;
        this.length = length;
        this.value = value;
    }

    public int getType() {
        return type;
    }

    public int getLength() {
        return length;
    }

    public byte[] getValue() {
        return value;
    }
}

```

```

        public boolean isConstructed() {
            return (tag & DerParser.CONSTRUCTED) ==
DerParser.CONSTRUCTED;
        }

        /**
         * For constructed field, return a parser for its
content.
         *
         * @return A parser for the construct.
         * @throws IOException
         */
        public DerParser getParser() throws IOException {
            if (!isConstructed())
                throw new IOException("Invalid DER: can't
parse primitive entity"); //$NON-NLS-1$

            return new DerParser(value);
        }

        /**
         * Get the value as integer
         *
         * @return BigInteger
         * @throws IOException
         */
        public BigInteger getInteger() throws IOException {
            if (type != DerParser.INTEGER)
                throw new IOException("Invalid DER: object is
not integer"); //$NON-NLS-1$

            return new BigInteger(value);
        }

        /**
         * Get value as string. Most strings are treated
         * as Latin-1.
         *
         * @return Java string
         * @throws IOException
         */
        public String getString() throws IOException {

            String encoding;

            switch (type) {

                // Not all are Latin-1 but it's the closest
thing

                case DerParser.NUMERIC_STRING:
                case DerParser.PRINTABLE_STRING:
                case DerParser.VIDEOTEX_STRING:
                case DerParser.IA5_STRING:
                case DerParser.GRAPHIC_STRING:
                case DerParser.ISO646_STRING:
                case DerParser.GENERAL_STRING:

```

```

        encoding = "ISO-8859-1"; //$NON-NLS-1$
        break;

    case DerParser.BMP_STRING:
        encoding = "UTF-16BE"; //$NON-NLS-1$
        break;

    case DerParser.UTF8_STRING:
        encoding = "UTF-8"; //$NON-NLS-1$
        break;

    case DerParser.UNIVERSAL_STRING:
        throw new IOException("Invalid DER: can't
handle UCS-4 string"); //$NON-NLS-1$

        default:
            throw new IOException("Invalid DER: object
is not a string"); //$NON-NLS-1$
        }

    return new String(value, encoding);
}
}

```


Anexo B

Código da extensão Chrome
ScriptProxy

Popup.html

```
<!doctype html>
<html>
<link href="./popup.css" type="text/css" rel="stylesheet">
<body>
  <h1>Javascript Signature Validator</h1>
  <h2>
    This extension is currently scanning if the Javascript
    present in the HTML pages you access are from a trusted source.
    In order to disable it, go to definitions -> extensions,
    and deactivate it.
  </h2>
</body>
</html>
```

Popup.css

```
body {
    width: 300px;
    height: 150px;
    margin: 10px 10px 10px;
    background-image:
url("http://www.downloadsources.com.br/upload/Screeny/The%20
Matrix%20Screensaver1.png");
}

h1 {
    color: white;
    font: 14px/1.2 Helvetica, sans-serif;
    font-size: 150%;
    margin: 20px 20px 10px;
    text-shadow: green 0 1px 2px;
}

h2 {
    color: white;
    font: 10px/1.2 Helvetica, sans-serif;
    font-size: 90%;
    margin: 20px 20px 10px;
    text-align: justify;
```

```
text-justify: inter-word;
}
```

Content.js

```
var isValid = false;
var i = 0;

//validação do JS
if(getHTML().indexOf("<script") >= 0){

    var count = (getHTML().match(/<script/g) ||
[]).length;

    while(true){

        var form = "formSig".concat(i);
        var formJs = form.concat(" .js");

        if($('#'+form).length &&
$('#'+formJs).length){

            //content do script
            var sMsg = $('#'+formJs).html().trim();
            //assinatura
            var hSig = $('#siggenerated'+i).val();
            //certificado
            var cert = $('#cert'+i).val();

            alert(cert);

            isValid = doVerify(sMsg, hSig, cert);

        }else{
```

```

        if(count > i+1){
            //se existirem scripts sem o form com a
            assinatura, o js, não é válido
            isValid = false;
        }

        break;
    }
    i = i+1;
}

}else{
    isValid = true;
}

function getHTML(){
    return document.documentElement.outerHTML
}

function doVerify(sMsg, hSig, cert) {

    //carrega o certificado e ve se a assinatura corresponde
    ao content no script
    var x509 = new X509();
    x509.readCertPEM(cert);
    var isValid = x509.subjectPublicKeyRSA.verifyString(sMsg,
hSig);

    return isValid;
}

//depois de validado, se todo o js nao for válido, pergunta
se quer continuar, senao volta para a página anterior
if(!isValid){

```

```
        if(!confirm("Javascript not signed! Continue anyway?")){
            history.go(-1);
        }
    }else{
        alert("Pls proceed!");
    }
}
```

Manifest.json

```
{
    "name": "Javascript Signature Validator",
    "version": "1",
    "description": "This extension checks that the web-pages are digitaly signed by a trusted CA.",
    "default_locale": "en",
    "browser_action": {
        "default_icon": "icon16.png",
        "default_popup": "popup.html"
    },
    "icons": {
        "16": "icon16.png"
    },
    "content_scripts": [
        {
            "matches": ["http://*/*", "https://*/*"],
            "run_at": "document_end",
            "js": [
                "jquery-1.11.3.min.js",
                "jsbn.js",
                "jsbn2.js",
                "rsa.js",
                "rsa2.js",
                "base64.js",
                "yahoo-min.js",
            ]
        }
    ]
}
```

```
    "core.js",
    "md5.js",
    "sha1.js",
    "sha256.js",
    "ripemd160.js",
    "x64-core.js",
    "sha512.js",
    "rsapem-1.1.js",
    "rsasign-1.2.js",
    "asn1hex-1.1.js",
    "x509-1.1.js",
    "crypto-1.1.js",
    "content.js"
  ]
}
],
"permissions": [
  "proxy"
],
"manifest_version": 2
}
```