UNIVERSIDADE FEDERAL DE PERNAMBUCO

Graduação em Ciência da Computação

Centro de Informática

2012.2

Análise do Aumento da Superfície de Ataque na Web Decorrente da Exploração de Recursos do HTML5

Trabalho de Graduação

Discente: Marcelo Frota Pinto Pessoa – mfpp@cin.ufpe.br

Orientador: Ruy José Guerra Barretto de Queiroz – ruy@cin.ufpe.br

Recife, Abril de 2013

Universidade Federal de Pernambuco

Graduação em Ciência da Computação

Centro de Informática

2012.2

Análise do Aumento da Superfície de Ataque na Web Decorrente da Exploração de Recursos do HTML5

Marcelo Frota Pinto Pessoa

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Ruy José Guerra Barretto de Queiroz

Assinaturas

Análise do Aumento da Superfície de Ataque na Web Decorrente da Exploração de Recursos do HTML5

_____

Ruy José Guerra Barretto de Queiroz

Orientador

_____

Marcelo Frota Pinto Pessoa

Discente

DON’T PANIC

– Douglas Adams

Agradecimentos

Agradeço primeiramente aos meus pais, Paulo e Vera, que me deram apoio incondicional em toda a minha vida, e que também não me deixaram na mão durante estes quase cinco anos de vida acadêmica. Se não fosse por eles, eu (literalmente) não estaria onde estou. Agradeço à minha irmã, Amanda, por ter sido uma grande amiga durante todos esses anos. Gostaria de agradecer à minha namorada (e futura esposa), Djamária, que foi minha principal incentivadora na finalização deste trabalho. Sem ela ao meu lado, este trabalho, provavelmente, não teria ficado pronto a tempo. Por fim, gostaria de prestar meus agradecimentos também a todos os companheiros da Tempest, onde aprendi praticamente tudo que sei sobre segurança da informação, e a meu orientador, Prof. Ruy de Queiroz, que foi de grande importância para minha formação.

Resumo

A introdução de novas funcionalidades, além de tornar uma aplicação potencialmente mais poderosa, torna-a também potencialmente mais vulnerável. O motivo é que o aumento da complexidade, quando se trata de segurança, inevitavelmente insere novos potenciais vetores de ataque. E com a chegada do HTML5 (*Hypertext Markup Language* versão 5) não será diferente, suas novas *features* ou introduzem novas vulnerabilidades ou tornam mais crítico o impacto de vulnerabilidades já conhecidas, causando assim o aumento da superfície de ataque na *web*, e tornando não só as aplicações, mas também os *browsers*, mais suscetíveis a ataques. Dessa forma, este trabalho visa realizar uma análise da insegurança das tecnologias que compõem o HTML5, com o objetivo de identificar novos vetores de ataque, e comprovar o aumento da superfície de ataque na *web*. Além disso, sempre que possível, este trabalho apresentará contramedidas para os ataques descritos.

Palavras-chave: HTML5, Segurança, Superfície de ataque, *Web*.

Abstract

The introduction of new features, apart from making an application potentially more powerful, it also makes it potentially more vulnerable. The reason is that the increase of complexity, when it comes to security, inevitably inserts new potential attack vectors. And with the arrival of HTML5 (Hypertext Markup Language version 5) is no different, its new features or introduce new vulnerabilities or make the impact of known vulnerabilities more critical, thus causing the increase of attack surface on the web, and making not only applications, but also web browsers, more susceptible to attacks. Thus, this work aims to conduct an analysis of the insecurity of the technologies that make up HTML5, with the aim of identifying new attack vectors, and prove the increase of attack surface on the web. In addition, whenever possible, this paper presents countermeasures for attacks described.

Keywords: *HTML5, Security, Attack surface, Web.*

Lista de figuras

Lista de tabelas

Lista de quadros

Sumário

1. Introdução

Este capítulo tem o propósito de introduzir o contexto da pesquisa realizada, ilustrando a importância desta para futuros trabalhos relacionados, além de apresentar os objetivos específicos deste trabalho e a forma como este documento foi estruturado.

1.1. Contexto e motivação

Não é novidade para ninguém o fato de que a Internet, mais precisamente a *web*, está presente nas mais diversas áreas, e que grande parte das tarefas executadas hoje, de alguma forma, a utiliza. A *web* está tão presente na vida das pessoas que até mesmo seus relacionamentos sociais estão dentro da rede, ou no mínimo, são influenciados por ela. Dada tamanha importância, as chamadas aplicações *web* vêm se tornando, cada vez mais, um alvo dos *hackers*. Os ataques sobre aplicações *web* tem registrado um aumento significativo nos últimos anos, com um destaque especial para as mais populares, como Facebook, Twitter, e MySpace, por exemplo [1]. Segundo estudos realizados pela Imperva [2], uma aplicação *web* comum é alvo de ataques, em média, 120 dias por ano (33% do tempo), enquanto que as mais visadas, encontram-se sob ataque em torno de 292 dias por ano (cerca de 80% do tempo).

Outro dado bastante preocupante, é que a maioria das aplicações *web* atuais possuem falhas de segurança, sejam elas no *front* ou no *back-end*. De acordo com a WhiteHat [3], o número de pontos suscetíveis a vulnerabilidades de alta severidade por aplicação é de 79 a cada ano. Este número, apesar de estar em queda, não é nada animador, visto que apenas uma vulnerabilidade dessa natureza pode ser o suficiente para que um eventual atacante consiga assumir o controle da aplicação. E com a chegada do **HTML5** (*Hypertext Markup Language* versão 5), que engloba um conjunto de novas tecnologias, essa situação tende a piorar.

A introdução de novas funcionalidades, além de tornar uma aplicação potencialmente mais poderosa, torna-a também potencialmente mais vulnerável. O motivo é que o aumento da complexidade, quando se trata de segurança, inevitavelmente insere novos potenciais vetores de ataque, causando o **aumento da superfície de ataque**. E com o HTML5 não será diferente, suas novas *features* ou introduzem novas vulnerabilidades ou tornam mais crítico o impacto de vulnerabilidades já conhecidas, causando assim o aumento da superfície de ataque na *web*, e tornando não só as aplicações, mas também os *browsers*, mais susceptíveis a ataques [4]. Dessa forma, surge a necessidade da realização de uma análise detalhada destas novas tecnologias por parte dos especialistas em segurança, de modo que seja possível estar em pé de igualdade com os *hackers*.

1.2. Objetivos

O objetivo principal deste trabalho é realizar uma análise da insegurança das tecnologias que compõem o padrão HTML5, com o intuito de identificar novos vetores de ataque, e assim comprovar o aumento da superfície de ataque na *web*. Além disso, como objetivo secundário, mas não menos importante, este trabalho tentará, sempre que possível, ilustrar formas de mitigar as ameaças apresentadas.

A metodologia empregada para a realização desta análise consistiu de quatro passos básicos, são eles:

1) Identificar os principais recursos que representam potenciais ameaças;
2) Apresentar as vulnerabilidades exploráveis nestes recursos;
3) Descrever possibilidades de ataque para cada recurso identificado;
4) Sugerir, quando possível, contramedidas para os ataques descritos.

1.3. Estrutura do trabalho

Este trabalho está organizado em cinco capítulos. O primeiro é o capítulo de introdução, e tem como objetivo apresentar o contexto, a importância, e os objetivos da pesquisa realizada. O segundo capítulo apresenta o conceito de superfície de ataque, e discorre sobre o aumento desta no decorrer do processo de evolução da *web*, até a chegada do HTML5. No terceiro capítulo, são identificados os principais recursos do HTML5 que configuram potenciais ameaças. O quarto capítulo apresenta possibilidades de ataque que exploram os recursos identificados no capítulo anterior, além de sugerir, quando possível, contramedidas para tais ataques. Por fim, no quinto capítulo, são apresentadas as conclusões sobre este trabalho, bem como sugestões de possíveis trabalhos futuros.

2. A superfície de ataque na web

Este capítulo tem como objetivo apresentar a definição de superfície de ataque que será adotada para os fins deste trabalho, bem como, ilustrar o crescimento da superfície de ataque na *web*, no decorrer de seu processo evolutivo. Apresentando, ao longo do processo, as questões de segurança levantadas, desde o surgimento da *web*, até a chegada do HTML5. Além disso, este capítulo também tem como objetivo apresentar o que é o HTML5 propriamente dito.

2.1. O que é superfície de ataque (na web)

Segundo Miguel Pupo Correia e Paulo Jorge Sousa [1], denomina-se de superfície de ataque, o conjunto de interfaces através das quais pode advir um ataque contra o sistema.

Para o projeto OWASP (*The Open Web Application Security Project*) [13], o termo superfície de ataque refere-se a todos os diferentes pontos através dos quais um atacante pode invadir o sistema, ou, pelo menos, extrair dados desse sistema.

Embora ambas as definições sejam satisfatórias para definir a superfície de ataque de um sistema, nenhuma delas é capaz de exprimir, com exatidão, a ideia de superfície de ataque na *web*. Dessa forma, com um olhar um pouco mais abrangente, e baseando-se nas duas definições apresentadas, pode-se definir a **superfície de ataque na web** da seguinte forma:

> "A superfície de ataque na *web* refere-se ao conjunto de todos os pontos ou vetores através dos quais um atacante é capaz de realizar um ataque, seja ele contra uma aplicação, ou contra um usuário".

2.2. A superfície de ataque no decorrer da evolução da web

Desde o seu surgimento, a *web* encontra-se em constante evolução, e no decorrer dessa evolução, à medida que novas tecnologias são incorporadas, novos vetores de ataque surgem, e novas questões de segurança são levantadas [12]. A seguir, é apresentado um breve resumo da evolução da *web*, desde o seu surgimento, até a chegada do HTML5 [14]:

- **1990 / 1991 – Surgimento da web (HTTP e HTML)**

Nesse momento a *web* é bastante simples, basicamente composta pelo protocolo HTTP e pelo HTML, e, obviamente, ainda não é o alvo preferido dos *hackers*.

- **1994 – Cookies**

Com o surgimento dos *cookies* questões relacionadas à privacidade e à segurança são levantadas, visto que através deles seria possível, por exemplo, obter informações sobre determinado usuário, o que permitiria, inclusive, identificá-lo.

- **1995 / 1996 – Java e Javascript**

A introdução dos *Applets* Java, e principalmente, do Javascript, desencadeia um processo sem volta. A partir desse momento o número de aplicações *web* começa a aumentar, e, paralelamente, vulnerabilidades como *Cross-Site Scripting* (XSS) passam a existir, implicando num aumento considerável da superfície de ataque na *web* (que antes era praticamente zero).

- **2000 – XHTML**

Nesse momento, o número de aplicações *web* já é razoavelmente grande, e o uso do XHTML torna-as ainda mais flexíveis, permitindo o seu acesso através de outros dispositivos, como telefones celulares e *palm tops*, por exemplo.

- **2004 / 2005 – AJAX, XHR e DOM**

A partir desse momento, diversos vetores de ataque são introduzidos, muitos deles silenciosos e imperceptíveis, para um usuário comum. O número de ataques contra aplicações *web*, e seus usuários, cresce rapidamente – tornando-as o alvo preferido dos *hackers*. Vetores de ataque recentes e antigos são combinados para a execução de ataques cada vez mais sofisticados, muitos deles incluindo etapas de engenharia social.

- **2008 – HTML5**

Neste ano, a especificação do HTML5 é, pela primeira vez, publicada na Internet, pelo WHATWG (*Web Hypertext Application Technology Working Group*). Vale ressaltar que, mesmo antes do lançamento de sua especificação, implementações parciais de suas *features* já existiam em alguns *browsers* – o que é bem perigoso, visto que tal tipo de implementação pode conter inúmeras falhas. Além disso, como mencionado brevemente no capítulo um, o HTML5 traz consigo uma série de novas tecnologias e recursos, configurando assim uma superfície completamente nova a ser explorada.

Cada etapa na evolução na *web* tem o seu próprio impacto na segurança do todo, e com o HTML5 não será diferente, novas ameaças estão surgindo no horizonte, e devem ser levadas a sério [12].

2.3. HTML5: uma nova superfície a ser explorada

O HTML5 não é apenas uma nova versão do HTML, ele compreende uma série de novos recursos e tecnologias que representam um novo modelo de arquitetura da *web*, permitindo o desenvolvimento de aplicações mais poderosas [15]. Ele foi projetado, principalmente, com o objetivo de tornar mais poderosa a linguagem de marcação vigente (neste caso, o HTML4) – melhorando o suporte à multimídia, a comunicação com o servidor, e tornando tarefas que antes eram bastante custosas, tanto em esforço como em desempenho, muito mais simples, deixando o trabalho do desenvolvedor muito mais fácil [15].

Como mencionado, o HTML5 é uma pilha de novos recursos e tecnologias, que aumenta a capacidade dos *browsers*, e permite a construção de aplicações *web* mais poderosas, dentro de um novo modelo de arquitetura *web*. Dentre as tecnologias e recursos introduzidos, pode-se citar: *Cross-Origin Resource Sharing*, *Client-Side Storage*, *Web Sockets*, *Web Workers*, *Web Messaging*, *Geolocation*, *Offline Application Caching*, etc., além da adição de novos elementos e atributos à linguagem HTML. Vale ressaltar ainda que, para permitir que a pilha em questão fosse, de fato, implementada, foi necessário estender alguns componentes básicos da arquitetura da *web* atual, como o *XMLHttpRequest* (XHR) e o *Document Object Model* (DOM), além de modificar a maneira como os *browsers* interpretam as *tags* HTML [12].

A disponibilização desses novos recursos, além das modificações realizadas em componentes já existentes, nos fornece uma superfície completamente nova a ser explorada. Ao tornar a arquitetura da *web* mais poderosa (e complexa), o HTML5 também a torna potencialmente mais vulnerável, inserindo novos pontos ou vetores, através dos quais é possível realizar diversos tipos de ataque – muitos deles difíceis de detectar, e altamente viáveis [12] – caracterizando, dessa forma, o aumento da superfície de ataque na *web*.

3. Novos recursos que representam potenciais ameaças

Como já mencionado nos capítulos anteriores, os novos recursos e tecnologias introduzidos pelo HTML5 implicam no aumento da superfície de ataque na *web*. E, de modo a ter uma visão mais palpável dessa superfície, este capítulo tem como propósito identificar os principais recursos do HTML5 que representam potenciais ameaças, explicando de forma resumida o funcionamento desses recursos, e por que eles representam uma ameaça, do ponto de vista de segurança.

A identificação de tais recursos não foi uma tarefa fácil, no entanto, tendo como base os trabalhos publicados por dois grandes pesquisadores da área: Michael Schmidt e Shreeraj Shah, foi possível definir – de maneira satisfatória – os cinco recursos do HTML5 que representam um maior nível de ameaça, e que, por conseguinte, serão analisados neste trabalho. São eles:

1) *Cross-Origin Resource Sharing* (CORS);
2) *Client-Side Storage* ("*Offline Storage*");
3) *Web Messaging*;
4) Novos elementos e atributos;
5) *Web Workers*.

3.1. Cross-Origin Resource Sharing (CORS)

3.1.1. Política de mesma origem (Same Origin Policy)

Com o intuito de prevenir ações maliciosas contra os usuários, ou outros *websites*, os *browsers* aplicam regras de restrição para requisições advindas do lado cliente de outras "origens". Uma "origem" é definida pela tripla composta pelos seguintes elementos: **protocolo**, **host**, **porta** [9].

O objetivo de tal proteção implementada nos *browsers* – conhecida como **política de mesma origem** (*Same Origin Policy*, ou SOP) – é restringir como um documento ou *script* carregado de uma determinada origem pode interagir com um recurso pertencente a outra origem. O comportamento padrão definido por essa política, é impedir o acesso a recursos por meio de scripts carregados de outras origens [10].

3.1.2. Descrição

Como já mencionado, por padrão, a política de mesma origem impede a realização de requisições entre origens diferentes (*Cross-Origin Request*, ou COR) através de mecanismos do lado cliente (como, Javascript, por exemplo). Tal comportamento é bastante problemático para aplicações *web* compostas por partes que acessam dados de outras origens. Como forma de obter tais dados, por exemplo, as aplicações precisam enviar uma requisição para o seu lado servidor, e este, por sua vez, realiza uma requisição ao domínio externo – solicitando os dados desejados – que por fim são transmitidos ao *script* rodando no cliente. Essa técnica, conhecida como *Server-Side Proxying*, além de comprometer o desempenho da aplicação, torna a sua implementação desnecessariamente complicada [4].

A partir dessa necessidade, foram desenvolvidas formas de relaxar a política de mesma origem, sendo o **Cross-Origin Resource Sharing (CORS)** uma forma legítima para tal [10].

O CORS torna possível o envio de *XMLHttpRequests* entre origens diferentes, através de um novo cabeçalho HTTP – denominado **Access-Control-Allow-Origin** – definido no recurso que se deseja compartilhar. Dessa forma, uma aplicação pode requisitar dados de outra origem, diretamente através de Javascript, sem ajuda do servidor [4].

Quando uma requisição *cross-origin* é realizada, o *browser* encarrega-se de decidir o que vai acontecer. No caso de uma requisição *cross-origin* simples, decide-se se a resposta obtida será, ou não, acessada através de Javascript. Entende-se como requisição *cross-origin* simples, uma requisição que utilize apenas os métodos GET, POST, ou HEAD; que altere apenas os cabeçalhos *Accept*, *Accept-Language*, ou *Content-Language*; e que tenha como *Content-Type*, um dos seguintes valores: *application/x-www-form-urlencoded*, *multipart/form-data*, ou *text/plain* [11].

Quando uma requisição desse tipo é realizada, o *browser* adiciona nesta um cabeçalho denominado **Origin**, que tem como função indicar a origem da requisição realizada [11]. A seguir é apresentado um exemplo desse cabeçalho:

```
Origin: http://minha.origem.com
```

O recurso a ser compartilhado possui uma lista de origens cujo acesso é permitido, e ao receber uma requisição, a aplicação – no lado servidor – verifica se a origem informada encontra-se nessa lista [11]. Em caso positivo, o cabeçalho *Access-Control-Allow-Origin* é preenchido com a origem enviada anteriormente, conforme exemplificado a seguir:

```
HTTP/1.1 200 OK
Content-Type: text/html
Access-Control-Allow-Origin: http://minha.origem.com
```

Também é possível que o cabeçalho em questão seja preenchido com um "*" (asterisco), quando o recurso encontra-se acessível para qualquer origem. No caso de a origem informada na requisição não possuir acesso ao recurso em questão, o cabeçalho *Access-Control-Allow-Origin* não é enviado na resposta [11].

Dessa forma, a partir da resposta obtida, o *browser* decide se o conteúdo da mesma pode ser lido pelo Javascript, ou não.

No caso em que a requisição *cross-origin* não é simples, isto é, não atende aos requisitos citados anteriormente, fica a cargo do *browser*, decidir se a requisição será, ou não, realizada. O procedimento executado é ligeiramente diferente do anterior, em vez de enviar a requisição original diretamente à aplicação, o *browser* realiza, antes disso, uma requisição prévia (*preflight request*), utilizando o método OPTIONS [11]. O objetivo dessa requisição prévia é verificar se o método e os cabeçalhos (se houver), utilizados na requisição original, são permitidos. E somente em caso positivo, é permitida a realização da requisição em questão [11].

O exemplo abaixo ilustra o comportamento do *browser* quando uma requisição PUT é realizada para o *host* **outra.origem.com** – que não permite o uso deste método em requisições *cross-origin* – a partir da seguinte origem: **http://minha.origem.com**.

Primeiramente, o *browser* realiza uma requisição prévia, como mencionado anteriormente, utilizando o método OPTIONS:

```
OPTIONS / HTTP/1.1
Host: outra.origem.com
Origin: http://minha.origem.com
Access-Control-Request-Method: PUT
```

Em seguida, o servidor responde ao *browser* quais métodos são permitidos:

```
HTTP/1.1 200 OK
Access-Control-Allow-Methods: GET,POST
Access-Control-Allow-Origin: *
```

Por fim, apesar de o recurso em questão estar disponível para qualquer origem (como indicado pelo "*"), o método PUT não se encontra entre os métodos permitidos pelo *host*, e por isso a requisição original é abortada pelo *browser*. Se o método PUT fosse permitido, a requisição original seria realizada normalmente.

3.1.3. Potenciais ameaças

Devido ao relaxamento da política de mesma origem, a introdução deste recurso pode representar uma grande ameaça, do ponto de vista de segurança. Como já mencionado, o CORS torna possível a realização de requisições *cross-origin* através de Javascript, de forma completamente silenciosa, isto é, sem o conhecimento do usuário [4]. Tal comportamento faz deste recurso um excelente vetor para a realização de ataques remotos, principalmente quando a lista de origens permitidas pela aplicação é excessivamente permissiva.

Um dos problemas fundamentais deste recurso consiste no fato de que, um *script* rodando no lado cliente, ou seja, no *browser* do usuário, é capaz de manipular as respostas das requisições realizadas. Dessa forma, uma página maliciosa criada por um eventual atacante, por exemplo, seria capaz de, através do abuso da sessão de um usuário, subverter o controle de acesso de aplicações [4]. Além disso, tal comportamento torna um ataque conhecido como *Cross-Site Request Forgery* (CSRF), potencialmente mais perigoso [12].

Outro problema relacionado ao CORS consiste no fato de a origem das informações não estar mais restrita à origem da própria aplicação. Dessa forma, tanto a origem, como os dados transmitidos através dessas requisições, não podem mais ser considerados confiáveis, e, por conseguinte, precisam ser validados apropriadamente no lado cliente – isto é, na aplicação que está requisitando tais dados – antes de serem exibidos ao usuário, por exemplo [4].

De forma mais específica, é possível afirmar que o *Cross-Origin Resource Sharing* fornece os vetores necessários para a realização de diversos tipos de ataque, os quais são apresentados na seção **4.1.** deste trabalho.

3.2. Client-Side Storage (“Offline Storage”)

3.2.1. Descrição

O termo “**Client-Side Storage**” é uma forma geral de se referir às diferentes, mas relacionadas, APIs integrantes do HTML5, que possibilitam o armazenamento de dados na máquina do usuário, em vez de no servidor [5]. Existem dois motivos principais que justificam o desenvolvimento de tais APIs:

- Tornar possível o uso da aplicação mesmo quando *offline*;
- Melhorar o desempenho das operações realizadas.

Diferentemente dos *cookies*, as informações armazenadas no *Client-Side Storage* não são enviadas para o servidor em toda requisição realizada, dessa forma, o espaço de armazenamento disponível para este recurso pode ser bem maior que o espaço disponibilizado para os *cookies*. O espaço de armazenamento pode variar de 5MB a 25MB – para cada API disponível – ou pode ser, inclusive, ilimitado, dependendo do *browser* utilizado [5]. No momento, existem três APIs para o armazenamento local de dados, são elas:

1) *Web Storage*;
2) *Web SQL Database*;
3) *Indexed Database*.

Vale ressaltar que, o acesso às informações armazenadas no *Client-Side Storage* (independente da API utilizada) é sempre restrito à “origem” que as armazenou, e que a manipulação desses dados é realizada, exclusivamente, através de Javascript [5].

3.2.2. APIs de armazenamento

Como já mencionado, existem três APIs distintas que possibilitam o armazenamento local de dados. Tais APIs são apresentadas, resumidamente, a seguir.

- **Web Storage**

O *Web Storage* configura um modelo de armazenamento simples, onde os dados são armazenados no formato chave-valor, ambos *Strings* [6]. Esta API fornece duas áreas distintas de armazenamento, são elas:

1) *Local Storage*;
2) *Session Storage*.

Os dados colocados no *Local Storage* podem ser manipulados por toda a origem que as armazenou, e persistem na máquina do usuário por quanto tempo for necessário, mesmo após o *browser* ser fechado. Os dados colocados no *Session Storage*, por sua vez, somente podem ser manipulados pela janela/aba à qual estão associados, e persistem na máquina do usuário apenas durante o tempo de vida dessa janela/aba, ou seja, até ela ser fechada [6].

Tanto o *Local* quanto o *Session Storage* implementam uma mesma interface, denominada **Storage**, e assim, compartilham dos mesmos atributos e métodos [6]. A interface Storage é apresentada a seguir:

```
interface Storage {
  readonly attribute unsigned long length;
  DOMString? key(unsigned long index);
  getter DOMString? getItem(DOMString key);
  setter creator void setItem(DOMString key, DOMString value);
  deleter void removeItem(DOMString key);
  void clear();
};
```

- **Web SQL Database**

O *Web SQL Database* configura um modelo de armazenamento estruturado, com toda a funcionalidade – e complexidade – de um típico SGBD relacional baseado em SQL. Esta API garante eficiência e praticidade na realização de consultas, e, além disso, fornece suporte a **transactions** – garantindo a consistência dos dados armazenados. As implementações atuais baseiam-se exclusivamente na biblioteca **SQLite** (sendo este um dos motivos pelos quais sua especificação encontra-se paralisada) [7].

O *Web SQL Database* fornece dois modos de uso: o **assíncrono** e o **síncrono**. O modo síncrono foi desenvolvido para ser utilizado apenas pelos *Web Workers*. O modo assíncrono, por sua vez, funciona com ou sem *Web Workers*, e seus métodos são implementados, de forma geral, através das interfaces apresentadas a seguir:

```
      typedef sequence<any> ObjectArray;

      interface SQLTransaction {
        void executeSql(in DOMString sqlStatement, in optional
ObjectArray arguments, in optional SQLStatementCallback callback, in
optional SQLStatementErrorCallback errorCallback);
      };

      [Callback=FunctionOnly, NoInterfaceObject]
      interface SQLStatementCallback {
        void handleEvent(in SQLTransaction transaction, in
SQLResultSet resultSet);
      };

      [Callback=FunctionOnly, NoInterfaceObject]
      interface SQLStatementErrorCallback {
        boolean handleEvent(in SQLTransaction transaction, in SQLError
error);
      };
```

- **Indexed Database**

O *Indexed Database* (ou *IndexedDB*) configura um modelo de armazenamento chave-valor avançado, onde o valor associado a uma chave pode assumir, inclusive, o formato de um **Javascript Object**. Dessa forma, os pares chave-valor funcionam como "registros", e são armazenados em estruturas denominadas **Object Stores** – que correspondem às tabelas do modelo relacional. Esta API também permite a criação de **índices**, baseados nas propriedades dos "registros" armazenados, permitindo a realização de consultas mais eficientes [8]. Pode-se dizer que, o *IndexedDB* é uma tentativa de agregar os pontos fortes das duas APIs anteriores, unindo simplicidade, eficiência e consistência. Vale ressaltar que, tal API também fornece suporte a **transactions** – garantindo a consistência dos dados armazenados.

O *IndexedDB* fornece dois modos de uso: o **assíncrono** e o **síncrono**. O modo síncrono foi desenvolvido para ser utilizado apenas pelos *Web Workers*. O modo assíncrono, por sua vez, funciona com ou sem *Web Workers*, e seus métodos são implementados, de forma geral, através da interface apresentada a seguir:

```
      interface IDBObjectStore {
        readonly attribute DOMString name;
        readonly attribute DOMString keyPath;
        readonly attribute DOMStringList indexNames;
        readonly attribute IDBTransaction transaction;
        readonly attribute boolean autoIncremenent;
        IDBRequest put(any value, optional any key);
        IDBRequest add(any value, optional any key);
        IDBRequest delete(any key);
        IDBRequest get(any key);
        IDBRequest clear();
        IDBRequest openCursor(optional any? range, optional DOMString
direction);
        IDBIndex createIndex(DOMString name, any keyPath, optional
IDBIndexParameters optionalParameters);
        IDBIndex index(DOMString name);
        void deleteIndex(DOMString indexName);
        IDBRequest count(optional any key);
      };
```

3.2.3. Potenciais ameaças

A maior preocupação, no que concerne à segurança, consiste no fato de o usuário não ter conhecimento do tipo de informação que é armazenada em sua máquina. Além disso, o usuário não tem controle sobre o acesso às informações armazenadas no *Client-Side Storage*, tal controle é exercido pela aplicação que as armazenou [4].

Como já mencionado, a manipulação dos dados é realizada através de Javascript, tornando-se transparente para o usuário. E, apesar de o acesso aos dados ser permitido, estritamente, ao domínio de origem, um atacante poderia – através de vulnerabilidades como *Cross-Site Scripting* (XSS) – ser capaz de roubar e/ou adulterar tais dados [4].

A presença do *Client-Side Storage* pode fornecer os vetores necessários para a realização de diversos tipos de ataque, os quais são apresentados na seção **4.2.** deste trabalho.

3.3. Web Messaging

3.3.1. Descrição

As aplicações *web* desenvolvidas necessitam, cada vez mais, incluir componentes (*widgets*) criados por terceiros. Esses componentes são, em sua maioria, mini aplicações em Javascript que possuem propósitos específicos, como fornecer informações sobre o clima, notícias, etc. No HTML4, existem apenas duas opções para incluir tais componentes dentro de uma aplicação: (a) através de *iframes* ou (b) através da inclusão de *scripts* de fontes externas que rodam tais componentes [4].

A primeira opção é mais segura, no entanto, o isolamento imposto pelo uso de *iframes* impede que a aplicação se comunique com componentes carregados de outra origem, fazendo com que o usuário tenha que informar, novamente, determinadas informações que já havia informado antes, o que impacta na usabilidade da aplicação como um todo [4].

A segunda opção é mais poderosa, visto que o Javascript incluído é capaz de acessar as informações presentes no DOM na aplicação, permitindo assim a comunicação entre a aplicação e seus componentes. No entanto, a inclusão de *scripts* de fontes externas é uma prática bastante perigosa, visto que falhas de segurança na fonte externa poderão se propagar, e afetar diretamente a aplicação [4].

O **Web Messaging**, introduzido pelo HTML5, tem como objetivo prover uma forma usável e segura de utilizar componentes em uma aplicação *web*, permitindo o compartilhamento de informações entre documentos, mesmo que eles não pertençam à mesma origem. Através desse recurso uma aplicação pode comunicar-se diretamente com qualquer componente carregado através de *iframes*, e, além disso, também é possível a comunicação dos *iframes* entre si [4].

A troca de mensagens implementada neste recurso baseia-se em eventos, de modo que cada mensagem representa um evento instanciado a partir da interface **MessageEvent**, definida a seguir [16]:

```
      [Constructor(DOMString type, optional MessageEventInit
eventInitDict)]
      interface MessageEvent : Event {
        readonly attribute any data;
        readonly attribute DOMString origin;
        readonly attribute DOMString lastEventId;
        readonly attribute (WindowProxy or MessagePort)? source;
        readonly attribute MessagePort[]? ports;
      };

      dictionary MessageEventInit : EventInit {
        any data;
        DOMString origin;
        DOMString lastEventId;
        WindowProxy? source;
        MessagePort[]? ports;
      };
```

Os atributos mais relevantes de um "evento de mensagem" são os atributos *data* e *origin*. O primeiro representa o valor da mensagem que está sendo transmitida, e o segundo representa a origem que enviou a mensagem.

3.3.2. Modos de comunicação

A especificação do *Web Messaging* define ainda dois modos de comunicação distintos: o **Cross-Document Messaging** e o **Channel Messaging** [16]. Detalhes sobre ambos os modos são apresentados a seguir.

- **Cross-Document Messaging**

O *Cross-Document Messaging* permite a comunicação entre documentos de diferentes origens, por exemplo, a troca de mensagens entre uma *webpage* e um *iframe* cujo conteúdo é carregado a partir de outra origem qualquer. O envio das mensagens em questão é realizado através do método **postMessage()**, que para este modo de comunicação exige a passagem de dois argumentos [17]:

1) *message*: a mensagem que se deseja enviar;
2) *targetOrigin*: a origem que receberá a mensagem.

O valor do argumento *message* não está limitado apenas a *Strings*. Objetos do Javascript, e outros tipos de dados mais interessantes, como *FileList*, *File Blob* ou *ArrayBuffer*, por exemplo, também podem ser enviados como mensagens [16].

O argumento *targetOrigin*, por sua vez, pode receber apenas os seguintes valores: uma URL absoluta – indicando uma origem específica, um asterisco (“*”) – indicando qualquer origem, ou uma barra (“/”) – indicando a mesma origem do emissor da mensagem [16].

A seguir é ilustrado um exemplo da troca de mensagens realizada entre uma *webpage* – cuja origem é **http://minha.origem.com** – e um *iframe* dessa mesma *webpage*, cujo conteúdo é carregado a partir de **http://outra.origem.com**.

Abaixo é apresentado o código HTML/Javascript presente na *webpage* que enviará a mensagem, por exemplo, **http://minha.origem.com/emissor.html**.

```
...
<iframe id="iframe" src="http://outra.origem.com/dest.html"></iframe>

<script>
    var dest = document.getElementById('iframe');
    dest.contentWindow.postMessage('ping', 'http://outra.origem.com');
</script>
...
```

Abaixo é apresentado o código HTML/Javascript presente na *webpage* que é carregada num *iframe* a partir do endereço **http://outra.origem.com/dest.html**, e que, neste caso, será o receptor da mensagem enviada**.**

```
...
<script>
    window.addEventListener('message', receiver, false);

    function receiver(e) {
        if (e.origin == 'http://minha.origem.com') {
           if (e.data == 'ping') {
               e.source.postMessage('pong', e.origin);
           } else {
               alert('Mensagem não esperada...');
           }
        }
    }
</script>
...
```

Como pode ser observado no exemplo, ao receber um evento de mensagem, a *webpage* “dest.html” verifica a origem (*origin*) do emissor, o valor (*data*) da mensagem, e, caso ambos contenham os valores esperados, envia uma resposta ao emissor da mensagem.

- **Channel Messaging**

O *Channel Messaging* fornece os meios necessários para o estabelecimento de um canal de comunicação direta, e em duas vias, entre contextos independentes (por exemplo, entre *iframes*, entre *scripts*, etc.) de uma mesma *webpage* [17].

Este modo de comunicação é indicado para o caso em que se deseja realizar uma comunicação direta, por exemplo, entre dois *iframes* de uma *webpage*. Tal comunicação também poderia ser feita utilizando a *webpage* que contém esses *iframes* como "*proxy*", através do modo *Cross-Document Messaging*. No entanto, não seria uma boa ideia, pois nesse caso a *webpage* teria que confiar cegamente em todas as mensagens transmitidas (o que é bastante perigoso), ou então verificar cada uma delas (o que atrapalharia o desempenho da comunicação) [17].

Por isso, de forma a facilitar esse processo, o *Channel Messaging* permite o estabelecimento de um **canal** (*MessageChannel*) de comunicação direto entre dois *iframes*, definindo cada um deles como uma **porta** (*MessagePort*) desse canal [17].

A forma de utilização deste modo de comunicação é muito semelhante a do *Cross-Document Messaging*, e, por isso, não será demonstrada com exemplos.

3.3.3. Potenciais ameaças

O *Web Messaging* certamente torna mais fácil e mais segura, a integração de uma *webpage* com componentes (*widgets*) carregados a partir de fontes externas. No entanto, como toda a comunicação dá-se no *client-side*, validações implementadas no lado servidor não surtirão qualquer efeito. Dessa forma, o uso negligente deste recurso pode representar uma séria ameaça à segurança da aplicação, principalmente quando a aplicação manipula informações sensíveis.

Algumas das potenciais ameaças relacionadas ao uso negligente do recurso *Web Messaging* são apresentadas na seção **4.3.** deste documento.

3.4. Novos elementos e atributos

3.4.1. Descrição

O HTML5 adiciona uma série de **novos elementos e atributos** à linguagem HTML, através dos quais é possível realizar ações que antes não eram possíveis apenas com o uso do HTML [12].

O quadro a seguir ilustra alguns dos novos elementos introduzidos:

Quadro 1 – Alguns elementos introduzidos pelo HTML5

Elemento	Descrição
<audio>	Define um conteúdo de som.
<video>	Define um vídeo ou filme.
<source>	Define os recursos de mídia para <video> e <audio>.
<article>	Define um artigo.
<section>	Define uma seção do documento.
<canvas>	Usado para criar desenhos, através do uso *scripts*.

O quadro a seguir, por sua vez, ilustra alguns dos novos atributos introduzidos:

Quadro 2 – Alguns atributos introduzidos pelo HTML5

Atributo	Descrição
autofocos	Determina que o elemento em questão deve receber o foco do cursor quando a página for carregada.
sandbox	Aplica uma série de restrições ao conteúdo carregado num <iframe>, por exemplo, impedir que o conteúdo carregado execute *scripts* no *browser* do usuário.
formaction	É utilizado para sobrescrever o atributo "action" de um <form>.

3.4.2. Potenciais ameaças

Os novos elementos e atributos adicionados ao HTML, embora não introduzam diretamente novos vetores de ataque, contribuem para o aumento das chances de sucesso de um atacante, ao realizar ataques já conhecidos, como *Cross-Site Scripting* (XSS) e *Clickjacking* [15]. E, dessa forma, diversas aplicações *web* que não se encontravam vulneráveis a tais tipos de ataque, por exemplo, podem agora estar vulneráveis, caso suas defesas não tenham sido implementadas da melhor forma possível.

Na seção **4.4.** deste trabalho, são apresentadas algumas possibilidades de ataque relacionadas ao ponto levantado.

3.5. Web Workers

3.5.1. Descrição

Existem várias questões que impedem a portabilidade de aplicativos interessantes, que rodam no lado servidor, para o Javascript, que roda no lado do cliente. Uma dessas questões é o fato de o Javascript ser *single-thread*, o que significa que não é possível executar vários *scripts* ao mesmo tempo [19].

Atualmente, de modo a contornar tal obstáculo, os desenvolvedores fazem uso de determinadas técnicas que "simulam" o paralelismo, como AJAX (*Asynchronous Javascript and XML*), por exemplo. No entanto, nenhuma das técnicas disponíveis representa uma forma real de paralelismo, fazendo com que múltiplas tarefas sejam executadas de forma satisfatória, mas não ideal [19].

É nesse contexto que o recurso **Web Workers** é introduzido. A especificação desse recurso define uma API que permite a execução de *scripts* (Javascript) em segundo plano, independente de qualquer *script* que esteja rodando na interface com o usuário [18]. O uso de *Workers* (como são chamados esses *scripts* que rodam em segundo plano) permite que tarefas longas sejam executadas paralelamente, sem que sejam interrompidas por outros *scripts* rodando ou por ações do usuário, e ainda, sem "bloquear" o *browser* do usuário [18]. Para alcançar o paralelismo, os *Workers* se comportam como se fossem *threads* de uma linguagem de alto nível [19].

A troca de mensagens realizada entre eles, da mesma forma que no recurso *Web Messaging*, baseia-se em eventos, de modo que cada mensagem representa um evento instanciado a partir da interface **MessageEvent** – já apresentada na seção **3.3.1.** deste documento. No entanto, diferentemente do *Web Messaging*, o envio de mensagens é realizado através de um novo método **postMessage()** – desta vez exclusivo para *Web Workers* – onde o valor da mensagem (que pode, inclusive, ser vazio) é passado no argumento *message* [18].

A título de ilustração, é apresentado a seguir um exemplo de um *Worker* simples, que apenas reflete a mensagem enviada pela *webpage* que o criou.

Abaixo é apresentado o código HTML/Javascript presente na *webpage* principal. Como pode ser observado, ela cria um *Worker* a partir do arquivo **workerSimples.js**, e, em seguida, envia-lhe uma mensagem com o valor "eco".

```
...
<p>Resposta: <span id="result"></span></p>

<script>
    var worker = new Worker('workerSimples.js');

    worker.onmessage = function(e) {
        document.getElementById('result').textContent = e.data;
    };

    worker.postMessage('eco');
</script>
...
```

Abaixo se encontra o código Javascript referente ao arquivo **workerSimples.js** (isto é, o *Worker*). Como pode ser observado, o *Worker* em questão apenas envia de volta a mesma mensagem recebida (neste caso, "eco").

```
...
self.onmessage = function(e) {
    self.postMessage(e.data);
};
...
```

Por fim, a *webpage* principal exibe ao usuário a resposta do *Worker*, inserindo-a no elemento cujo atributo "id" possui o valor "result".

3.5.2. Tipos de Web Workers

Existem dois tipos de *Workers*: os **dedicados** (*dedicated Workers*) e os **compartilhados** (*shared Workers*) [18]. Os *Workers* dedicados estão diretamente ligados ao seu criador, mas também podem se comunicar com outros componentes e outros *Workers* através de mensagens (*MessageEvents*) – convém ressaltar que, o exemplo apresentado no item **3.5.1.** ilustra um *Worker* dedicado. Os *Workers* compartilhados, por sua vez, são nomeados, dessa forma qualquer *script* rodando na mesma origem pode obter sua referência e se comunicar com ele.

3.5.3. Potenciais ameaças

Os *Web Workers* não introduzem, por si só, novos vetores de ataque, no entanto, podem potencializar ou simplificar a execução de outros ataques. Através do uso de *Web Workers*, por exemplo, torna-se bastante fácil construir uma *Botnet* de *browsers*, como elucidado por Lavakumar Kuppan em [20].

Na seção **4.5.** deste trabalho, são apresentadas algumas aplicações do recurso *Web Workers*, que podem beneficiar um eventual atacante.

4. Possibilidades de ataque e contramedidas

Este capítulo tem o propósito de apresentar quais são, exatamente, as ameaças relacionadas aos recursos identificados no capítulo anterior, isto é, quais são as possibilidades de ataque criadas, ou simplificadas, pelo uso dos cinco recursos definidos no capítulo três. Além disso, sempre que possível, serão sugeridas contramedidas para evitar, ou, pelo menos, dificultar a execução de tais ataques.

As possibilidades de ataque apresentadas encontram-se divididas de acordo com os recursos aos quais estão relacionadas, seguindo a mesma ordem adotada no capítulo três.

4.1. Cross-Origin Resource Sharing (CORS)

4.1.1. Acesso a aplicações de uma rede interna a partir da Internet

É possível acessar aplicações de uma rede interna, a partir de um ponto qualquer da Internet, desde que tais aplicações possuam páginas cujo cabeçalho *Access-Control-Allow-Origin* não esteja definido de maneira apropriada, isto é, esteja definido com o valor "*" (asterisco) [4]. Dessa forma, um atacante seria capaz de obter acesso a informações que apenas um usuário da rede interna deveria ter acesso, utilizando o *browser* desse usuário como "*proxy* ". O ataque em questão pode ser realizado facilmente, desde que o atacante consiga conduzir um usuário com acesso à rede interna a uma *webpage* maliciosa sob seu controle.

A figura a seguir ilustra a possibilidade de ataque descrita acima:

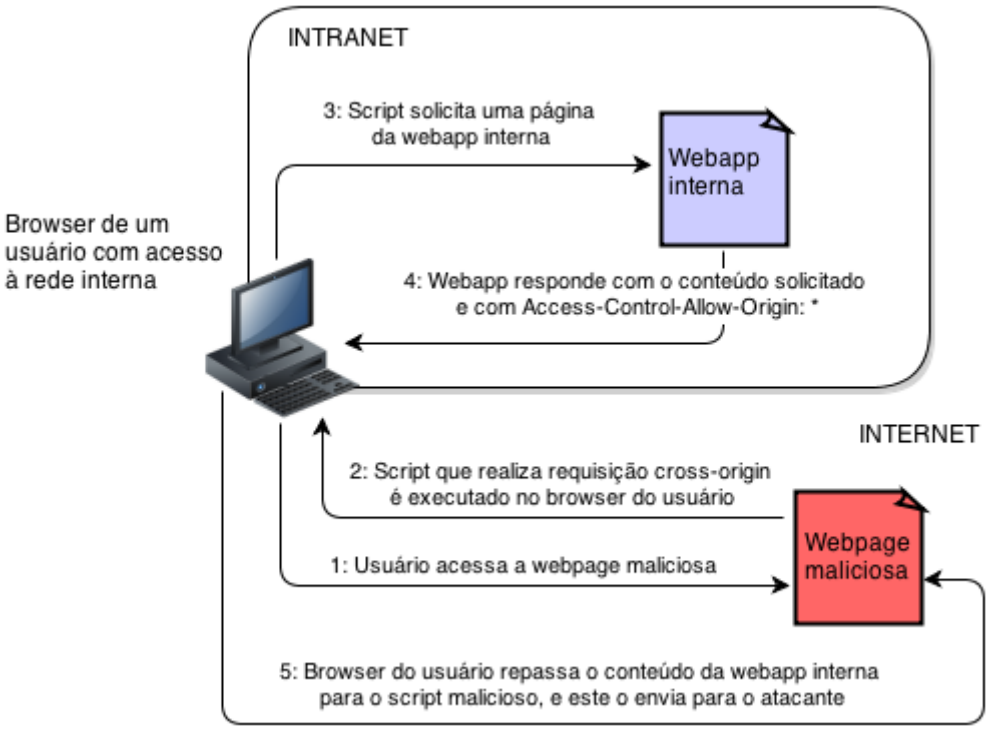

Figura 1 – Acesso a aplicações de uma rede interna a partir da Internet

- **Contramedidas**

Páginas cujo acesso é restrito a um determinado grupo, ou que contenham informações sensíveis, não devem ter o cabeçalho *Access-Control-Allow-Origin* definido com "*" (asterisco) [4]. Tal comportamento é considerado excessivamente permissivo, e deve ser evitado. Recomenda-se que a aplicação restrinja o acesso a tais páginas apenas às origens permitidas. Vale ressaltar ainda, que nunca se deve basear um controle de acesso, exclusivamente, no valor do cabeçalho *origin* (presente nas requisições *cross-origin*), visto que tal cabeçalho pode ter seu valor forjado, por exemplo, através de vulnerabilidades como *HTTP Header Injection* [4].

4.1.2. Varredura de uma rede interna baseada no tempo de resposta

É possível realizar a varredura de uma rede interna e identificar, por exemplo, os domínios existentes naquela rede, além de enumerar páginas pertencentes a estes domínios. Tudo isso através da análise do tempo de resposta das requisições *cross-origin* (*XMLHttpRequests*) realizadas, independente dela ter sido abortada pelo *browser*, ou não [4]. Da mesma forma que na seção **4.1.1.**, o ataque em questão pode ser realizado facilmente, desde que o atacante consiga conduzir um usuário com acesso à rede interna a uma *webpage* maliciosa sob seu controle. No entanto, para o sucesso deste ataque, o alvo nem mesmo precisa ter definido o cabeçalho *Access-Control-Allow-Origin* em suas páginas/recursos.

A figura a seguir ilustra um possível cenário para o ataque descrito acima:

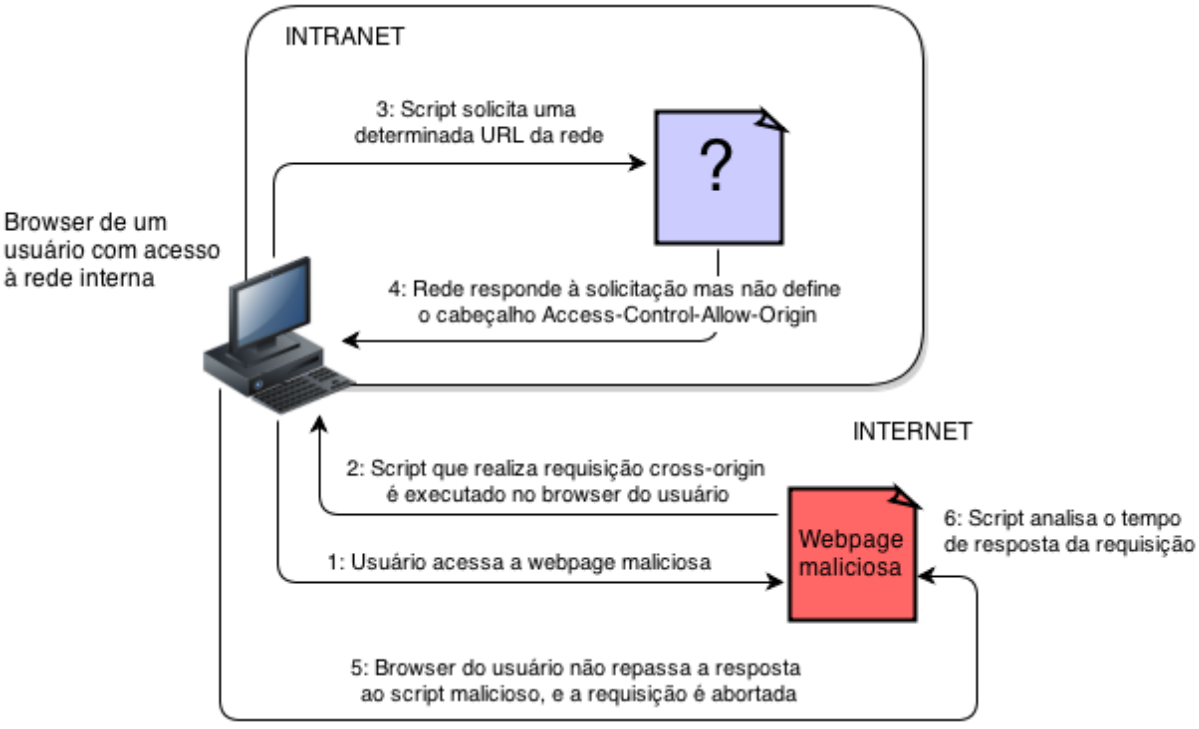

Figura 2 – Varredura de uma rede interna baseada no tempo de resposta

A tabela a seguir ilustra os resultados obtidos através da análise do tempo de resposta de alguns testes realizados:

Tabela 1 – Resultados da varredura baseada no tempo de resposta

Tempo de resposta	Interpretação
≈ 86 ms	O domínio solicitado não existe.
≈ 1170 ms	O domínio existe, mas a página solicitada não existe (HTTP 404).
≈ 1359 ms	Tanto o domínio, como a página solicitada, existem. No entanto, o acesso foi negado devido ao não envio do cabeçalho *Access-Control-Allow-Origin* na resposta.

- **Contramedidas**

Não existem meios viáveis de evitar o ataque descrito.

4.1.3. Realização de ataques remotos contra servidores e/ou aplicações

Como as requisições *cross-origin* simples são sempre realizadas, estas podem ser utilizadas para a realização de ataques remotos a servidores e/ou aplicações. Tal tipo de ataque pode ser executado, por exemplo, através do *browser* de um usuário qualquer, que por ventura acesse uma página contendo *scripts* maliciosos. Com um número suficientemente grande de usuários, acessando simultaneamente a página maliciosa, torna-se possível a realização de ataques de negação de serviço distribuídos, ou DDoS (*Distributed Denial of Service*) contra uma determinada aplicação [4]. Além disso, este ataque pode ainda ser potencializado pelo uso de *Web Workers*, conforme apresentado na seção **4.5.1.** deste trabalho. Vale ressaltar que, para o sucesso deste ataque, o alvo não precisa ter definido o cabeçalho *Access-Control-Allow-Origin* em suas páginas/recursos.

- **Contramedidas**

Não é possível evitar ataques remotos de forma geral, no entanto, é possível dificultar o sucesso de ataques de negação de serviço distribuídos (DDoS), conforme descrito nas contramedidas da seção **4.5.1.**

4.1.4. Estabelecimento de uma shell remota

Requisições *cross-origin* podem ser utilizadas para estabelecer uma *shell* remota no *browser* da vítima, e, dessa forma, controlar o comportamento do mesmo dentro de uma aplicação – desde que a aplicação alvo esteja vulnerável a *Cross-Site Scripting* (XSS). Nesta situação um atacante tem total controle da sessão da vítima, e utiliza o *browser* desta como se fosse um "*proxy*" para executar ações arbitrárias dentro de uma aplicação [4]. A grande vantagem deste tipo de ataque, quando comparado a um simples sequestro de sessão, é que o mesmo também funciona para aplicações acessíveis apenas por usuários de uma rede interna.

A possibilidade de ataque descrita pode ser demonstrada, por exemplo, através da ferramenta **Shell of the Future** [21], desenvolvida por Lavakumar Kuppan.

- **Contramedidas**

Se a aplicação alvo estiver, de fato, vulnerável a *Cross-Site Scripting* (XSS), não há meios de evitar o estabelecimento de uma *shell* remota no *browser* do usuário.

4.1.5. Potencialização de ataques de Cross-Site Request Forgery

O uso do CORS potencializa um vetor de ataque conhecido como *Cross-Site Request Forgery* (CSRF). Em primeiro lugar, porque torna possível a exploração deste vetor em duas vias, isto é, permite não só o envio da requisição forjada, como também a leitura da resposta (desde que o cabeçalho *Access-Control-Allow-Origin* esteja definido de forma que permita tal coisa) – o que permite a realização de ataques mais sofisticados. Em segundo lugar, porque permite o uso de *XMLHttpRequests*, inclusive, para o *upload* de arquivos (desde que a aplicação alvo, obviamente, possua algum formulário de *upload*) [12].

- **Contramedidas**

Não se aplicam, visto que, neste caso, o uso do CORS apenas potencializa a exploração de outro vetor de ataque.

4.2. Client-Side Storage ("Offline Storage")

4.2.1. Roubo de informações a partir de outras aplicações

Como não existe forma de restringir o acesso às informações armazenadas no *client-side* através do *pathname* (como se faz com os *cookies*, através do atributo *path*), aplicações hospedadas num mesmo domínio podem ter acesso às informações umas das outras [6][7][8]. Dessa forma, após subverter uma aplicação qualquer de um *host*, um atacante é capaz de roubar informações referentes a outras aplicações que estejam rodando em outros diretórios desse mesmo *host*. Tais ataques são conhecidos como *Cross-directory attacks*.

- **Contramedidas**

Não é possível impedir que informações armazenadas no *client-side* de uma aplicação sejam acessadas por outras que compartilhem do mesmo domínio. Dessa forma, recomenda-se que o recurso de armazenamento local (*Client-Side Storage*) não seja empregado em domínios compartilhados [4].

4.2.2. Novos vetores para ataques persistentes

Códigos maliciosos podem ser injetados, e armazenados no *client-side*, fornecendo novos vetores para ataques persistentes [4]. Tal situação poderia ser concretizada, por exemplo, aproveitando-se de um *Client-Side SQL Injection* – descrito na seção **4.2.6.** – ou através da exploração de um *Cross-Site Scripting* (caso a aplicação esteja vulnerável aos mesmos).

- **Contramedidas**

Não se aplicam, pois um método de armazenamento sempre estará sujeito à ocorrência de ataques persistentes (independentemente da forma com a qual foi desenvolvido/projetado), seja através de falhas de injeção, ou através de indivíduos mal intencionados que, por ventura, possam manipular os dados armazenados.

4.2.3. Roubo de informações sensíveis

Qualquer informação sensível (dados pessoais, credenciais de acesso, *tokens* de sessão, etc.) que venha a ser armazenada no *client-side* da aplicação, pode ser roubada, por exemplo, através da exploração bem sucedida de um *Cross-Site Scripting* (XSS), caso a aplicação esteja vulnerável a tal [6][7][8].

Mesmo que a aplicação não esteja suscetível a falhas de injeção, como XSS, ainda é possível executar o ataque em questão, desde que a vítima acesse a aplicação através de uma rede (uma *Wi-Fi* pública, por exemplo) que esteja sob a influência do atacante [4]. Neste caso, o atacante é capaz de interceptar e adulterar as respostas enviadas pela aplicação, adicionando a elas *scripts* maliciosos.

A figura a seguir ilustra o cenário de ataque em que a vítima acessa a aplicação através de uma rede *Wi-Fi* pública (sob controle do atacante):

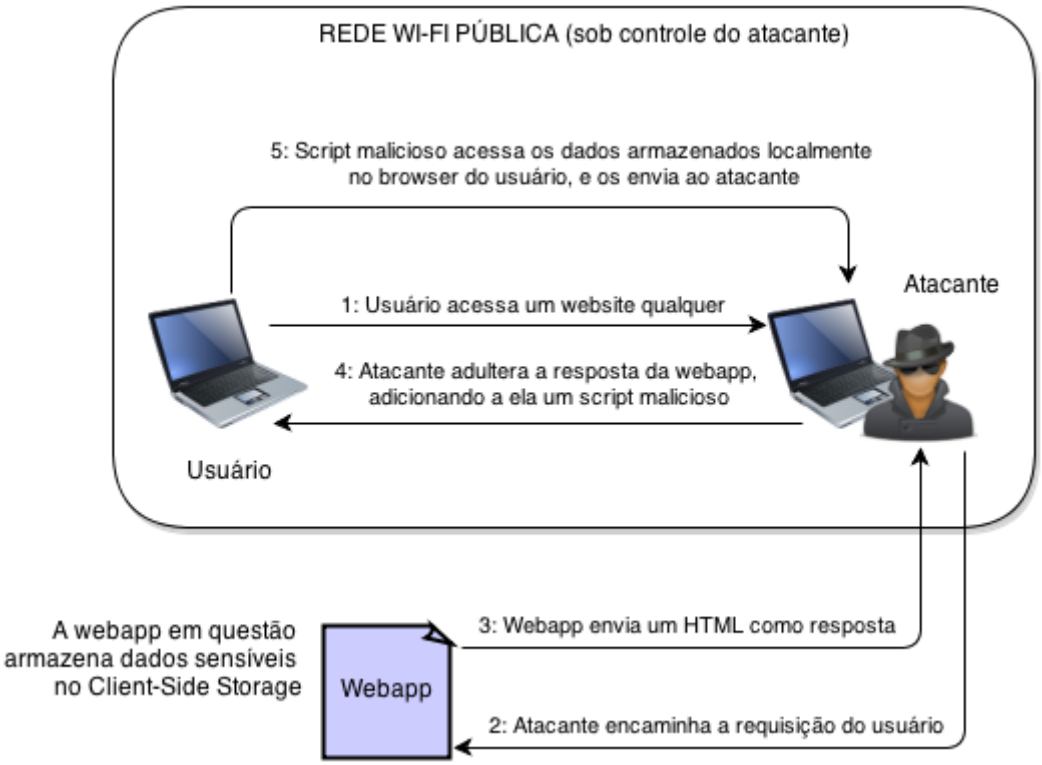

Figura 3 – Roubo de informações sensíveis armazenadas localmente

- **Contramedidas**

Não é possível impedir que as informações armazenadas no *client-side* sejam manipuladas através de Javascript, pois as APIs de armazenamento local foram projetadas para tal. Dessa forma, recomenda-se que nenhuma informação sensível seja armazenada no *Client-Side Storage* [4]. Em vez disso, tais informações devem ser armazenadas – exclusiva e adequadamente – no lado servidor [4].

4.2.4. Falsificação/adulteração de informações

Qualquer informação armazenada no *client-side* da aplicação pode ser falsificada/adulterada, por exemplo, através da exploração bem sucedida de um *Cross-Site Scripting* (XSS), caso a aplicação esteja vulnerável a tal [6][7][8]. Tal situação ocorre porque as APIs de armazenamento local foram projetadas para serem manipuladas via Javascript. Outra forma de executar o ataque em questão é aproveitando-se de um *Client-Side SQL Injection* – descrito na seção **4.2.6.** deste trabalho.

Vale ressaltar, ainda, que da mesma forma apresentada no item **4.2.3.**, o ataque em estão também pode ser executado em aplicações que não possuem falhas de injeção, desde que o usuário a acesse através de uma rede que esteja sob a influência do atacante.

- **Contramedidas**

Mesmo numa aplicação livre de falhas de injeção, não é possível garantir que as informações armazenadas no *client-side* não serão falsificadas/adulteradas. Dessa forma, tais informações não são confiáveis, e, portanto, não devem ser empregadas em operações/funcionalidades críticas.

4.2.5. Condições de corrida

Especificamente na API *Web Storage*, a manipulação de informações armazenadas no *client-side* está suscetível a condições de corrida (*Race Conditions*), visto que, a API em questão não oferece suporte a *transactions* [6]. Dessa forma, há grandes chances de os dados armazenados serem corrompidos durante a execução de operações concorrentes.

- **Contramedidas**

As condições de corrida mencionadas dizem respeito aos *browsers*, e não às aplicações *web*. Para evitá-las, os mantenedores de *browsers* precisam introduzir um semáforo global, no entanto, o uso deste pode reduzir bastante o desempenho das operações realizadas [6]. Alguns *browsers* implementam uma estrutura de semáforo chamada **Storage Mutex** [22] e eliminam a possibilidade de corrupção dos dados, no entanto, *browsers* como *Chromium* e *Opera*, por exemplo, preferem não sacrificar o desempenho, e dessa forma continuam suscetíveis a condições de corrida.

4.2.6. Injeção de comandos SQL no client-side

Especificamente na API *Web SQL Database*, a manipulação de informações armazenadas no *client-side* está suscetível a ataques de SQL *injection*, decorrente do mau uso do método *executeSql()* [7]. O método *executeSql()* permite inserir, de forma segura, os parâmetros fornecidos pelo usuário, em um comando SQL – através da substituição de *placeholders* indicados por uma "?" (interrogação) [7]. No entanto, como já mencionado, o uso negligente deste método deixa a aplicação suscetível a um ataque popularmente conhecido como *Client-Side SQL injection*.

O exemplo a seguir ilustra um *script* responsável por inserir um "*tweet*" em uma tabela denominada *tweet_db*, armazenada numa base de dados local – utilizando a API *Web SQL Database*.

```
tx.executeSql("INSERT INTO tweet_db (id, tweet, screenname) values
("+tweet.id+", '"+tweet.text+"', '"+tweet.user.screen_name+"')", []);
```

Observe que o método *executeSql()* é empregado de forma insegura, pois apesar de o mesmo permitir a realização de **consultas parametrizadas**, o desenvolvedor preferiu concatenar os valores de parâmetros controláveis pelo usuário ao comando SQL. O comportamento em questão permite que um atacante execute, de forma trivial, um ataque de injeção de comandos SQL no *client-side* (ou *Client-Side SQL injection*), com o intuito, por exemplo, de adulterar ou destruir os dados armazenados.

- **Contramedidas**

O ataque em questão pode ser evitado simplesmente fazendo um uso correto do método *executeSql()*, ou seja, utilizando consultas parametrizadas [7]. O exemplo a seguir ilustra o mesmo *script* de antes – responsável por inserir um "*tweet*" na tabela *tweet_db* – agora construído de maneira segura.

```
tx.executeSql("INSERT INTO tweet_db (id, tweet, screenname) values
(?,?,?)", [tweet.id, tweet.text, tweet.user.screen_name]);
```

Observe que os valores dos parâmetros controláveis pelo usuário não são concatenados diretamente, em vez disso, são passados como argumento para o método *executeSql()*, que ficará responsável por substitui-los, de maneira segura, nas posições ocupadas por "?", dentro do comando SQL executado.

4.3. Web Messaging

4.3.1. Vazamento de informação sensível

Ao enviar uma mensagem para um *iframe* através do método *postMessage()*, caso o argumento *targetOrigin* seja definido com um asterisco ("*"), as mensagens trocadas poderão ser "escutadas" por qualquer *iframe* presente na página, visto que a origem do receptor não foi restringida [16]. Dessa forma, um eventual atacante – capaz de inserir código Javascript na *webpage* (através da exploração de *Cross-Site Scripting*, por exemplo, ou sendo ele mesmo o provedor de um *widget*) – poderia, de maneira trivial, obter informações sensíveis que estejam sendo transmitidas nas mensagens, caracterizando o vazamento de informação [4].

A figura a seguir representa a situação descrita acima:

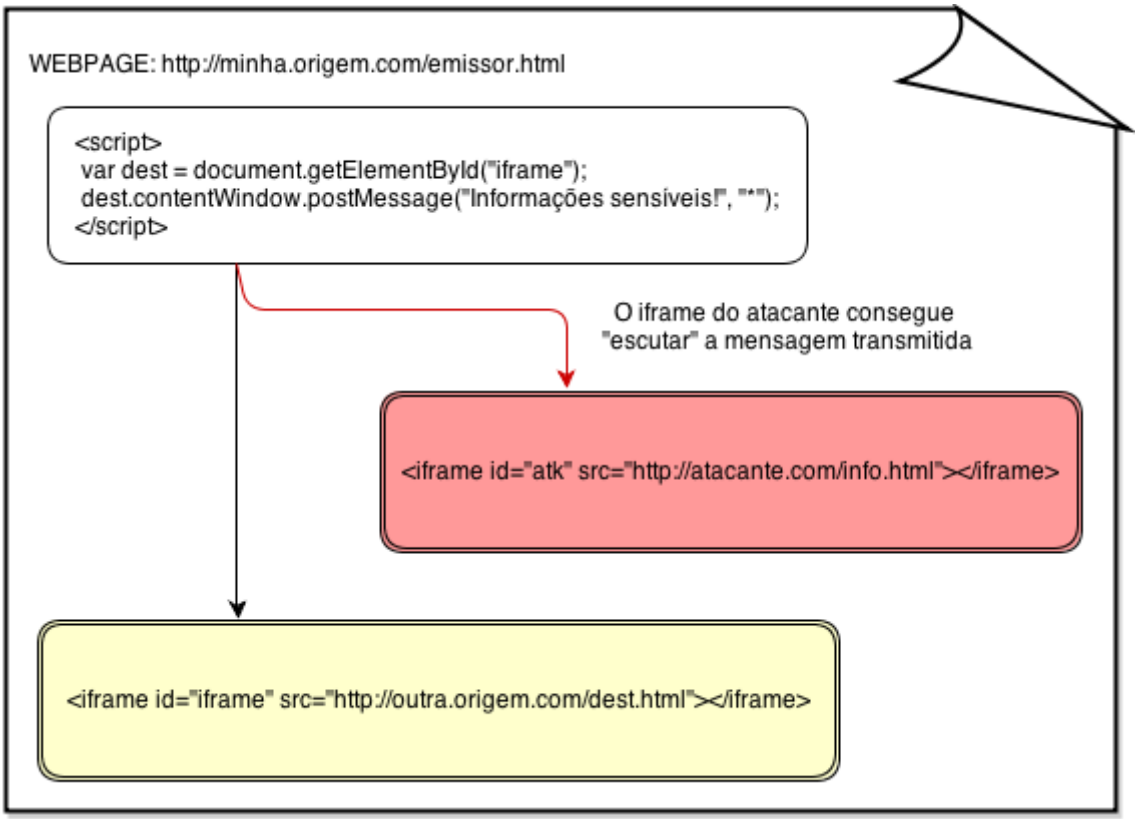

Figura 4 – Vazamento de informação sensível na troca de mensagens

- **Contramedidas**

No método *postMessage()*, deve-se definir o argumento *targetOrigin* com o valor exato da origem para qual se deseja enviar a mensagem [16]. Dessa forma, caso a mensagem transmitida contenha informações sensíveis, estas serão recebidas apenas pela origem alvo.

4.3.2. Possibilidade de ataques de negação de serviço

Aplicações que utilizam o recurso de *Web Messaging*, e aceitam mensagens oriundas de qualquer origem, isto é, não verificam o atributo *origin* da mensagem, podem estar sujeitas a ataques de negação de serviço (*DoS*), visto que mensagens enviadas por *websites* hostis, por exemplo, também serão processadas pela aplicação [16]. O ataque em questão torna-se viável, principalmente, quando a operação realizada após a chegada de uma mensagem é custosa.

- **Contramedidas**

Ao receber uma mensagem através de *Web Messaging*, o receptor deve tomar o cuidado de verificar adequadamente a origem do emissor (atributo *origin* da mensagem), de modo que apenas mensagens oriundas de origens esperadas sejam processadas pela aplicação [16].

4.3.3. Possibilidade de Cross-Site Scripting

As mensagens recebidas através de *Web Messaging* são oriundas de fontes externas, e por isso, podem não ser confiáveis [16]. Caso a aplicação não valide apropriadamente as informações recebidas, e as utilize diretamente como código HTML – através da propriedade *innerHTML*, por exemplo – ela estará sujeita a um ataque de *Cross-Site Scripting* específico, denominado *DOM-Based XSS*, permitindo que um eventual atacante execute código HTML/Javascript no *browser* do usuário [12].

- **Contramedidas**

Ao receber uma mensagem através de *Web Messaging*, o receptor deve tomar o cuidado de validar adequadamente o valor da mensagem recebida (atributo *data* da mensagem) [4]. Além disso, sempre que possível, deve-se evitar que as informações recebidas sejam utilizadas como *innerHTML* [4], ou que sejam passadas para determinadas funções do Javascript, reconhecidas como inseguras, como: *eval()*, *setInterval()* e *setTimeout()*.

4.4. Novos elementos e atributos

4.4.1. Possibilidade de Cross-Site Scripting

Grande parte dos novos elementos e atributos adicionados ao HTML permite a execução de código Javascript, e, dessa forma, aplicações que utilizam apenas listas negras (*blacklists*) como contramedida para ataques de *Cross-Site Scripting* (XSS), podem agora estar vulneráveis a tais ataques. A proteção em questão pode ser contornada através de ataques que utilizem elementos e/ou atributos que não sejam "conhecidos" pela lista negra definida [15].

A seguir são apresentadas novas possibilidades de executar Javascript no *browser* do usuário [12], e que podem ser empregadas em ataques de XSS:

```
1) <form><button formaction="javascript:alert('XSS')">Click
2) <math href="javascript:alert('XSS')">Click</math>
3) <math><maction actiontype=""
xlink:href="javascript:alert('XSS')">Click</maction></math>
4) <input autofocus onfocus=alert('XSS')>
5) <select autofocus onfocus=alert('XSS')>
6) <textarea autofocus onfocus=alert('XSS')>
7) <keygen autofocus onfocus=alert('XSS')>
8) <video><source onerror="javascript:alert('XSS')">
9) <audio><source onerror="javascript:alert('XSS')">
```

A imagem a seguir exemplifica a execução de um *Cross-Site Scripting* a partir de uma das formas listadas acima, utilizando o *browser* Mozilla Firefox:

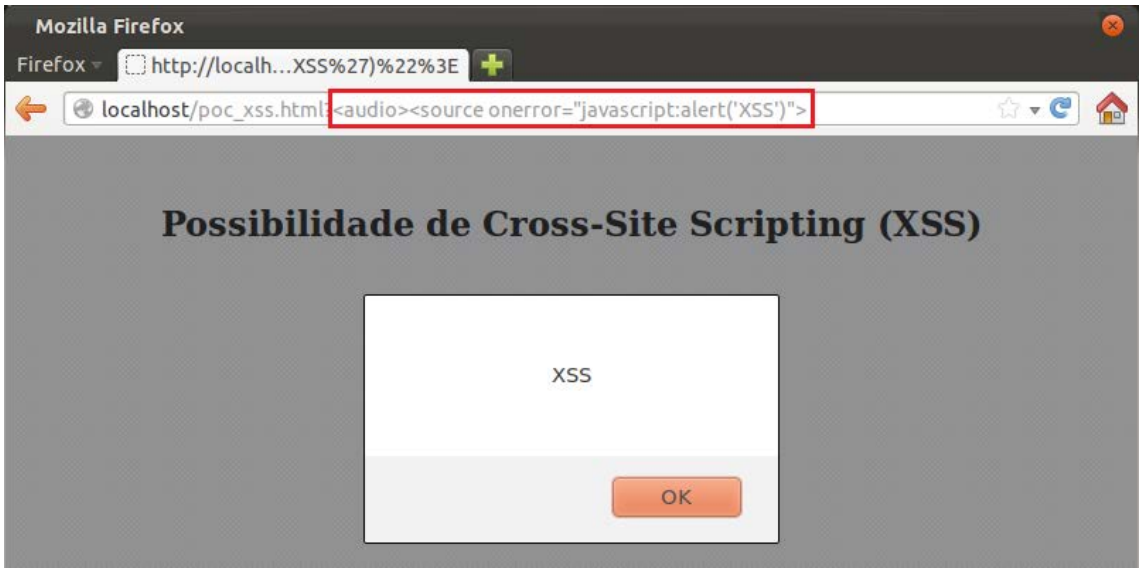

Figura 5 – Possibilidade de XSS utilizando novos elementos e atributos

O exemplo em questão ilustra apenas a execução da função *alert()* contendo a mensagem "XSS", no entanto, através de desse vetor de ataque, um atacante é capaz de executar o Javascript que quiser no *browser* do usuário. Na maioria dos casos, o objetivo é roubar os *cookies* do usuário.

- **Contramedidas**

A melhor forma de eliminar vulnerabilidades de injeção de HTML/Javascript é validar apropriadamente todas as entradas fornecidas pelo usuário, e codificar todas as saídas que serão exibidas a ele – utilizando HTML *entities*, por exemplo. No entanto, caso seja estritamente necessário o uso de listas negras (*blacklists*), deve-se garantir que todos os novos elementos e atributos "problemáticos" – isto é, que permitem a execução de Javascript no *browser* do usuário – sejam devidamente "bloqueados" pela lista negra definida.

4.4.2. Possibilidade de Clickjacking

O atributo *sandbox*, apesar de ter sido criado para aumentar a segurança ao carregar conteúdos não confiáveis através de *iframes*, em determinadas situações torna possível, a um atacante, realizar ataques de *Clickjacking* contra os usuários de aplicações que, a priori, não estariam vulneráveis [15]. Tal situação ocorre porque o atributo *sandbox* pode ser definido para impedir a execução de *scripts* pelo *website* carregado no *iframe*, e por tabela, acaba desativando uma das mais populares proteções contra *Clickjacking*, denominada *framekilling* ou *framebusting*, que é implementada em Javascript [15]. Dessa forma, uma aplicação que empregue apenas o *framekilling* como proteção contra tal ataque, estará, na verdade, vulnerável a ele.

- **Contramedidas**

Como forma de evitar ataques de *Clickjacking*, pode-se adicionar, às páginas da aplicação, uma medida declarativa de segurança denominada *X-Frame-Options* [25]. O *X-Frame-Options* é um novo cabeçalho HTTP, introduzido com o intuito de evitar que uma *webpage* seja "*frameada*", isto é, que seja carregada dentro de um *frame* (ou *iframe*). O cabeçalho em questão pode assumir três valores [25]:

1) *DENY*: impede que a página seja carregada dentro de qualquer *frame*;
2) *SAMEORIGIN***:** permite que a página seja carregada apenas em *frames* da mesma origem;
3) *ALLOW-FROM origin*: permite que a página seja carregada apenas em *frames* da origem especificada (pelo parâmetro *origin*).

Definindo o valor desse cabeçalho, a aplicação informa ao *browser* como ele deve proceder ao se deparar com situações em que se tenta carregar páginas dessa aplicação em *frames* (ou *iframes*).

4.4.3. Possibilidade de carregamento de conteúdo malicioso

O atributo *sandbox*, apesar de ser uma boa ideia, ainda não é interpretado por todos os *browsers* disponíveis. Por conta disso, qualquer aplicação que utilize esse atributo como única forma de proteção para seus *iframes*, pode estar vulnerável ao carregamento de conteúdos maliciosos – caso carregue conteúdos a partir de fontes externas. De forma geral, pode-se dizer que, caso o *browser* da vítima não "entenda" o atributo *sandbox*, torna-se possível executar código Javascript através de conteúdos maliciosos que, por ventura, sejam carregados em *iframes* da aplicação [4].

- **Contramedidas**

De forma geral, não é possível confiar em conteúdos carregados a partir de fontes externas, pois, caso a fonte em questão tenha sua segurança comprometida, ataques realizados a partir dela poderão se propagar para a aplicação. A presença do atributo *sandbox* certamente aumenta grau de segurança nessas situações, no entanto, como mencionado, ele não é compreendido por todos os *browsers* [4]. Dessa forma, recomenda-se evitar o carregamento de conteúdos de fontes externas – e, portanto, não confiáveis – em *iframes* da aplicação.

4.5. Web Workers

4.5.1. Ataques de negação de serviço distribuídos

Como já mencionado na seção **4.1.3.** deste documento, através de requisições *cross-origin* é possível executar ataques de negação de serviço distribuídos ou DDoS (*Distributed Denial of Service*) contra servidores e/ou aplicações, desde que uma quantidade suficientemente grande de usuários acesse uma página maliciosa que tenha tal objetivo. Com o suporte dos *Web Workers*, o número de *browsers* necessários para executar tal ataque cai drasticamente, pois a execução de *scripts* simultâneos permite que um único *browser* (que acesse a página maliciosa) realize cerca de 10.000 requisições por segundo, para o alvo do ataque [4].

A figura a seguir ilustra a possibilidade de ataque descrita acima:

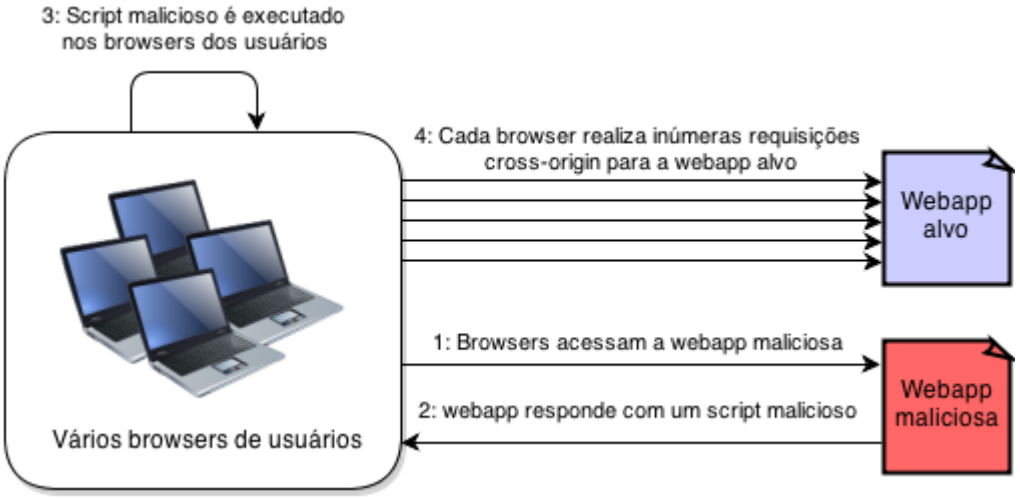

Figura 6 – DDoS utilizando CORS e Web Workers

- **Contramedidas**

Não é possível mitigar ataques de negação de serviço distribuídos ou DDoS (*Distributed Denial of Service*) através de desenvolvimento seguro. No entanto, é possível diminuir as chances de sucesso dessa categoria de ataque, através da utilização de *Web Application Firewall* (WAF). Para tal, regras de *firewall* devem ser configuradas para bloquear requisições CORS sempre que as mesmas chegarem com grande frequência à aplicação. Vale lembrar que as requisições CORS podem ser reconhecidas pelo cabeçalho *Origin*, adicionado a elas pelo *browser*.

4.5.2. Quebra de hashes criptográficos

Como qualquer linguagem de programação, o Javascript pode ser utilizado para executar ataques de força bruta com o objetivo de "quebrar" *hashes* criptográficos (como *hashes* MD5 ou *hashes* da família SHA), no entanto, o seu desempenho é, em geral, de 100 a 115 vezes mais lento que um código nativo rodando na mesma máquina [23]. Com o suporte dos *Web Workers*, essa desvantagem no desempenho é compensada pela distribuição do processamento entre várias *threads*, de diversos *browsers* – que acessem a página contendo os *scripts* para quebra de *hashes* [4]. Dessa forma, um eventual atacante seria capaz de "quebrar" *hashes* criptográficos de maneira rápida, sem a necessidade de possuir um servidor potente, e dedicado a esta tarefa.

A possibilidade descrita acima pode ser demonstrada, por exemplo, através da ferramenta **Ravan** [23], desenvolvida por Lavakumar Kuppan.

- **Contramedidas**

Não se aplicam.

4.5.3. Mineração de Bitcoins

Os *Web Workers* podem ser utilizados para auxiliar a mineração (ou geração) de *Bitcoins* [15], através da distribuição do processamento necessário entre várias *threads*, de diversos *browsers* – que acessem a *webpage* contendo o *script de* mineração.

Mineradores de *Bitcoins* totalmente desenvolvidos em Javascript, e utilizando *Web Workers*, já estão sendo desenvolvidos. Um belo exemplo de mineradores desse tipo é o **Bitcoin JavaScript Miner** [24].

- **Contramedidas**

  Não se aplicam.

5. Conclusão e trabalhos futuros

A análise realizada expõe, de forma clara, o aumento da superfície de ataque na *web* em decorrência da exploração dos novos recursos e tecnologias que compõem o HTML5. Novos vetores de ataque surgiram. Vetores já conhecidos, como *Cross-Site Scripting* (XSS), tornaram-se mais fáceis de explorar, e mais eficazes. Outros vetores de ataque, como *Cross-Site Request Forgery* (CSRF), foram potencializados, tornando-se mais perigosos. Percebe-se que o uso do HTML5 torna não só as aplicações *web*, mas também os *browsers*, mais susceptíveis a ataques.

São muitas as possibilidades de ataque conhecidas, e grande parte desses ataques não pode ser evitado através de codificação segura, o que é bastante preocupante. Mais preocupante do que isso, são as possibilidades de ataque ainda desconhecidas, e que podem surgir a qualquer momento.

Por fim, vale ressaltar que a proposta do WHATWG (*Web Hypertext Application Technology Working Group*) não é criar uma versão "redonda" para o padrão HTML, mas sim transformá-lo num padrão vivo (*Live Standard*), de modo que esteja em constante evolução e aprimoramento. Do ponto de vista de segurança, a proposta em questão pode representar sérios problemas, visto que, dessa forma, a superfície de ataque na *web* estaria em constante mudança (ou seja, em provável crescimento).

Em relação a trabalhos futuros, a análise realizada fornece a base necessária para uma análise mais aprofundada da insegurança na *web*. E, além disso, fornece também informações preciosas a cerca de novas vulnerabilidades decorrentes da exploração de recursos do HTML5, que podem ser utilizadas para incrementar as diversas ferramentas de varredura, denominadas *scanners*, existentes no mercado.

Glossário

Atacante – Indivíduo com conhecimento técnico suficiente, motivado, e com intenção de realizar ataques, seja contra aplicações ou contra usuários.

Hacker – Para os fins deste trabalho, o mesmo que atacante.

Cross-Site Scripting (XSS) – Ataque que permite a injeção e execução de código HTML/Javascript no *browser* de um usuário, a partir de falhas de validação presentes na aplicação.

DOM-Based XSS – Categoria de *Cross-Site Scripting* resultante de uma injeção de código diretamente no DOM (*Document Object Model*) interpretado no *browser* do usuário, não sendo necessária a passagem pelo servidor.

Cross-Site Request Forgery (CSRF) – Ataque que "pega carona" na sessão de um usuário autenticado numa aplicação alvo, com o objetivo de realizar ações nessa aplicação, como se fosse o próprio usuário. De forma geral, o ataque em questão acontece quando a vítima acessa uma página maliciosa, enquanto ainda está autenticado na aplicação escolhida como alvo do ataque.

Clickjacking – Ataque que tem como princípio enganar o usuário, fazendo-o clicar em algo diferente do que ele acha que está clicando. Tal tipo de ataque pode ser utilizado, dentre outras coisas, para roubar informações sensíveis do usuário.

Denial of Service (DoS) – Ataque de negação de serviço. Ataque que tem como objetivo tornar os recursos de um sistema indisponíveis para seus usuários.

Distributed Denial of Service (DDoS) – Ataque de negação de serviço distribuído. Ataque de DoS em grande escala, realizado de forma distribuída.

HTTP Header Injection – Ataque que permite a injeção de cabeçalhos HTTP, criados dinamicamente, a partir de parâmetros controláveis pelo usuário.

Bitcoin – Unidade monetária digital e anônima. Devido ao anonimato garantido, são muito utilizadas em transações financeiras ligadas a atividades criminosas.

Referências bibliográficas

[1] CORREIA, Miguel Pupo; SOUSA, Paulo Jorge. **Segurança no Software**. Lisboa: FCA, 2010.

[2] **Imperva's Web Application Attack Report**. Ed. 3, 2012. Disponível em: <http://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed3.pdf>. Acesso em: 27 dez. 2012.

[3] **WhiteHat's Website Security Statistics Report**. Ed. 12, 2012. Disponível em: <https://www.whitehatsec.com/assets/WPstats_summer12_12th.pdf>. Acesso em: 03 jan. 2013.

[4] SCHMIDT, Michael. **HTML5 Web Security**. Compass Security AG, 2011. Disponível em: <http://dl.packetstormsecurity.net/papers/web/HTML5_Web_Security_v1.0.pdf>. Acesso em: 02 dez. 2012.

[5] MAHEMOFF, Michael. **Client-Side Storage**. Disponível em: <http://www.html5rocks.com/en/tutorials/offline/storage/>. Acesso em: 02 fev. 2013.

[6] **Web Storage**. Disponível em: <http://dev.w3.org/html5/webstorage/>. Acesso em: 02 fev. 2013.

[7] **Web SQL Database**. Disponível em: <http://www.w3.org/TR/webdatabase/>. Acesso em: 03 fev. 2013.

[8] **Indexed Database API**. Disponível em: <http://www.w3.org/TR/IndexedDB/>. Acesso em: 03 fev. 2013.

[9] **Same Origin Policy**. Disponível em: <http://www.w3.org/Security/wiki/Same_Origin_Policy>. Acesso em: 07 fev. 2013.

[10] **Same Origin Policy**. Disponível em: <http://en.wikipedia.org/wiki/Same_origin_policy>. Acesso em: 07 fev. 2013.

[11] **Cross-Origin Resource Sharing**. Disponível em: <http://www.w3.org/TR/cors/>. Acesso em: 07 fev. 2013.

[12] SHAH, Shreeraj. **HTML5 Top 10 Threats Stealth Attacks and Silent Exploits**. In: BLACK HAT EUROPE, 2012, Netherlands. Disponível em: <http://media.blackhat.com/bh-eu-12/shah/bh-eu-12-Shah_HTML5_Top_10-WP.pdf>. Acesso em: 10 jan. 2013.

[13] **Attack Surface Analysis Cheat Sheet**. Disponível em: <https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet>. Acesso em: 15 fev. 2013.

[14] **The Evolution of the Web**. Disponível em: <http://www.evolutionoftheweb.com/?hl=en>. Acesso em: 16 fev. 2013.

[15] MCARDLE, Robert. **HTML5 Overview**: A Look at HTML5 Attack Scenarios. Trend Micro, 2011. Disponível em: <http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt_html5-attack-scenarios.pdf>. Acesso em: 15 jan. 2013.

[16] **HTML5 Web Messaging**. Disponível em: <http://dev.w3.org/html5/postmsg/>. Acesso em: 08 mar. 2013.

[17] **An Introduction to HTML5 Web Messaging**. Disponível em: <http://dev.opera.com/articles/view/window-postmessage-messagechannel/>. Acesso em: 11 mar. 2013.

[18] **Web Workers**. Disponível em: <http://dev.w3.org/html5/workers/>. Acesso em: 16 mar. 2013.

[19] **The Basics of Web Workers**. Disponível em: <http://www.html5rocks.com/en/tutorials/workers/basics/>. Acesso em: 16 mar. 2013.

[20] KUPPAN, Lavakumar**. Attacking with HTML5**. In: BLACK HAT ABU DHABI, 2010, United Arab Emirates. Disponível em: <http://media.blackhat.com/bh-ad-10/Kuppan/Blackhat-AD-2010-Kuppan-Attacking-with-HTML5-wp.pdf>. Acesso em: 11 jan. 2013.

[21] KUPPAN, Lavakumar**. Shell of the Future:** Reverse Web Shell Handler for XSS Exploitation. Disponível em: <http://blog.andlabs.org/2010/07/shell-of-future-reverse-web-shell.html>. Acesso em: 09 jan. 2013.

[22] **A vocabulary and associated APIs for HTML and XHTML**. Storage Mutex. Disponível em: <http://www.w3.org/TR/html5/webappapis.html#storage-mutex>. Acesso em: 02 abr. 2013.

[23] KUPPAN, Lavakumar**. Cracking hashes in the JavaScript cloud with Ravan**. Disponível em: <http://blog.andlabs.org/2010/12/cracking-hashes-in-javascript-cloud.html>. Acesso em: 09 jan. 2013.

[24] **A Bitcoin miner implemented in JavaScript**. Disponível em: <https://github.com/progranism/Bitcoin-JavaScript-Miner>. Acesso em: 02 abr. 2013.

[25] **The X-Frame-Options response header**. Disponível em: <https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options>. Acesso em: 05 abr. 2013.