Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Humberto Borgas Bulhões

Abordagens para mitigação de riscos de segurança da informação em navegadores: uma proposta baseada em JavaScript e HTML

São Paulo

2018

Humberto Borgas Bulhões

Abordagens para mitigação de riscos de segurança da informação em navegadores: uma proposta baseada em JavaScript e HTML

Exame de Qualificação apresentado ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, como parte dos requisitos para a obtenção do título de mestre em Engenharia de Computação.

Data da aprovação ____/____/_____

Prof. Dr. Marcelo Novaes de Rezende (Orientador)
IPT – Instituto de Pesquisas Tecnológicas do Est. de SP

Membros da Banca Examinadora:

Prof. Dr. Marcelo Novaes de Rezende (Orientador)
IPT – Instituto de Pesquisas Tecnológicas do Est. de SP

Prof. Dr. Alexandre José Barbieri de Sousa
Mestrado em Engenharia de Computação

Prof. Dr. Plínio Roberto Souza Vilela
Universidade Estadual de Campinas – UNICAMP

Humberto Borgas Bulhões

Abordagens para mitigação de riscos de segurança da informação em navegadores: uma proposta baseada em JavaScript e HTML

Exame de Qualificação apresentado ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, como parte dos requisitos para a obtenção do título de mestre em Engenharia de Computação.

Área de Concentração: Engenharia de Software

Orientador: Prof. Dr. Marcelo Novaes de Rezende

São Paulo

Janeiro/2018

Humberto Borgas Bulhões

Abordagens para mitigação de riscos de segurança da informação em navegadores: uma proposta baseada em JavaScript e HTML / Humberto Borgas Bulhões. São Paulo, 2018.

42 p. : il. (algumas color.); 30 cm.

Orientador Prof. Dr. Marcelo Novaes de Rezende

Dissertação de Mestrado – Instituto de Pesquisas Tecnológicas do Estado de São Paulo, 2018.

CDU XX:XXX:XXX.X

RESUMO

Os programas navegadores oferecem amplos recursos para a execução de aplicações voltadas para a web. Suas funcionalidades, porém, podem expor a informação dos usuários a riscos de vazamento e adulteração. Ainda que os navegadores venham sendo melhorados para que consigam detectar e mitigar alguns desses riscos, esse esforço é marcado por concessões às capacidades esperadas pelas aplicações web, fazendo com que a segurança da informação no navegador seja um campo de conhecimento com iniciativas e padrões consideradas incoerentes entre si. Por causa disso, desenvolvedores de aplicações nem sempre podem prever o grau de exposição dos dados de seus usuários. Seria importante que fosse possível determinar, de modo programável e imperativo, que esses dados ficassem fora do alcance de participantes não confiáveis, particularmente *scripts*. Para essa finalidade, este trabalho propõe uma abordagem que proporcione ao desenvolvedor uma barreira de proteção incorporada às aplicações. Nesta proposta, o desenvolvedor tem controle direto sobre a exposição das informações que considerar sensíveis, em contraste com o controle indireto, declarativo e tolerante a vazamento de informação empregado atualmente nos navegadores. A validação dessa proposta ocorrerá por meio de um protótipo de sistema submetido a situações de risco de vazamento da informação em páginas web, em correspondência com ocorrências documentadas pela literatura. O protótipo será preparado para que a abordagem proposta, sob a forma de um *script* baseado em APIs padronizadas de HTML e Javascript, seja colocada à prova no seu propósito de neutralizar tais situações de risco.

Palavras-chave: Segurança da informação, vazamento de dados, HTML, Javascript, DOM.

LISTA DE ILUSTRAÇÕES

Figura 1 – Componentes do navegador de código aberto Chromium 21
Figura 2 – Aplicação web composta por conteúdo proveniente de duas origens. 25
Figura 3 – Domínio de CDN comprometido, capturando informações do usuário. 26

LISTA DE CÓDIGOS

2.1 Vazamento de dados em fluxo explícito de informação 20
2.2 Vazamento de dados em fluxo implícito de informação 20
2.3 Página HTML incorporando *script* de outra origem 25
3.1 Determinação da visibilidade da *shadow tree* e sua consequência na tentativa de acesso ao DOM subjacente 35
3.2 Demonstração de herança prototípica em Javascript 36
3.3 Exemplo de método nativo interceptado por um script 37

LISTA DE TABELAS

Tabela 1 – Responsabilidades dos subsistemas do navegador 22

LISTA DE ABREVIATURAS E SIGLAS

CORS	Cross-Origin Resource Sharing
CSRF	Cross-Site Request Forgery
CSP	Content Security Policy
DAC	Discretionary Access Control
DOM	Document Object Model
IFC	Information Flow Control
JSON	JavaScript Object Notation
MAC	Mandatory Access Control
SOP	Same Origin Policy
SRI	Subresource Integrity
XSS	Cross-Site Scripting

SUMÁRIO

1 INTRODUÇÃO 11
1.1 Motivação 11
1.2 Objetivo 14
1.3 Contribuições 14
1.4 Método de trabalho 15
1.5 Organização do trabalho 16

2 FUNDAMENTOS 18
2.1 Principais conceitos 18
2.2 Conceitos básicos 18
2.2.1 Segurança da informação 18
2.2.2 Modelos de controle de acesso 19
2.2.3 Controle do fluxo de informações 19
2.2.4 Fluxos da informação no navegador 21
2.2.4.1 Ciclo de vida de uma página da web 23
2.2.5 Caracterização do problema 24
2.2.6 Vulnerabilidades 24
2.2.6.1 Compartilhamento do ambiente de execução 24
2.2.6.2 Cross-Site Scripting (XSS) 26
2.2.6.3 Comprometimento de extensões 27
2.2.7 Aparato de segurança da informação implementado pelos navegadores 27
2.2.8 Requisitos para avaliação da segurança da informação no navegador 27
2.2.9 Abordagens para mitigação de riscos 28
2.2.9.1 Políticas padronizadas 28
2.2.9.1.1 SOP – Same Origin Policy 28
2.2.9.1.2 CSP – Content Security Policy 29
2.2.9.1.3 CORS – Cross-Origin Resource Sharing 30
2.2.9.1.4 SRI – Subresource Integrity 30
2.2.9.2 Abordagens proprietárias 30
2.2.9.2.1 Abordagens experimentais 31
2.2.9.2.2 JSFLow 31
2.2.9.2.3 SessionGuard 31
2.2.9.2.4 Abordagens em processo de padronização 31
2.2.9.2.5 COWL 31
2.2.9.2.6 CSP Nível 3 31
2.3 Shadow DOM 31
2.4 Estado da arte 32
2.4.0.1 Security of web mashups: A survey (DeRyck et al., 2012) 32
2.4.0.2 Toward Principled Browser Security (YANG et al., 2013) 32
2.4.0.3 JSFlow: Tracking information flow in JavaScript and its APIs (HEDIN et al., 2014) 33
2.4.0.4 Protecting Users by Confining JavaScript with COWL (STEFAN et al., 2014) 33
2.4.0.5 Information Flow Control for Event Handling and the DOM in Web Browsers (RAJANI et al., 2015) 34
2.4.0.6 The Most Dangerous Code in the Browser (HEULE et al., 2015a) 34

3 PROPOSTA 35
3.1 API de ocultação do DOM fundamentada em Shadow DOM 35
3.1.1 Interceptação de métodos nativos do ambiente de execução 36
3.2 Casos de teste de exposição aos riscos de vazamento de informação no navegador 37
3.3 Apresentação do site acadêmico “jstest.me” 38
3.4 Roteiro para a elaboração da proposta 38

REFERÊNCIAS 39

1 INTRODUÇÃO

1.1 Motivação

O software navegador é um ponto de convergência da informação na web. Seus usuários dependem dele para trabalhar, estudar e facilitar tarefas, e também para a comunicação e o entretenimento. Essas aplicações são causa e consequência da evolução do *web browser*, que ao longo do tempo se transformou em uma plataforma para a distribuição de software. Hoje, muito distante de um simples navegador de conteúdo, o *browser* provê recursos capazes não apenas de capturar, transmitir e apresentar informação, mas também de executar programas (*scripts*) destinados ao enriquecimento da experiência do usuário e ao compartilhamento de informação.

O amplo escopo das funcionalidades do navegador, potencializado por sua utilização maciça, torna esse software um alvo para ataques à confidencialidade e à integridade da informação que circula pelas aplicações da web. FonteA indica que X% dos usuários já estiveram vulneráveis ao vazamento de informações pessoais durante o uso de um navegador, enquanto FonteB estima que cerca de N ataques à segurança da informação ocorrem durante (periodicidade). Em resposta, a comunidade de desenvolvimento dos *browsers* incorporou ao software diversos mecanismos para a defesa de seus usuários. Porém, problemas fundamentais, originadores de vulnerabilidades, permanecem intrínsecos às arquiteturas dos navegadores, perpetuando riscos.

Assim, o software navegador é vetor de uma variedade de ataques à segurança da informação. Tais ataques são fundamentados em tecnologias que, ao mesmo tempo em que dão poder aos desenvolvedores de aplicações web, também tornam possíveis o vazamento e adulteração de dados. Elementos de composição de página como Javascript, *frames*, aplicativos Java e conteúdo em Flash levam a experiência de usuário na web a um patamar mais elevado em termos de funcionalidade, porém mais perigoso no que diz respeito ao conjunto de vulnerabilidades aproveitáveis por fraudadores e *hackers*. Ademais, esses elementos de página também não foram concebidos com políticas de segurança claras e consistentes. Nessas circunstâncias, a segurança das aplicações web complexas se baseia mais na confiança que seus autores deposi-

tam em seus componentes do que em políticas de segurança coerentes estabelecidas pelos navegadores.

Recursos reiteradamente vulneráveis (cit?), aplicativos para Java e Flash acabaram em desuso, e os navegadores deixaram de oferecer suporte nativo a essas tecnologias (VERGE, 2016; ADOBE, 2017). Javascript, ao contrário, cresceu em funcionalidade e popularidade entre os desenvolvedores, impulsionando uma forma de aplicação web marcada pelo *conteúdo ativo*. O conteúdo ativo tira proveito de uma característica dos navegadores chamada de DOM (*Document Object Model*), uma interface de programação que permite a manipulação do conteúdo das páginas por *scripts*. É relevante destacar que *scripts* podem ser carregados pelo navegador a partir de múltiplas origens – *sites* diferentes daquele que hospeda a aplicação principal. Um exemplo desse tipo de conteúdo pode ser encontrado em *blogs* e *sites* jornalísticos, dentro dos quais podem ser incorporados anúncios provenientes de *sites* de serviços publicitários. Nesses casos, o navegador cria um contexto de execução compartilhado onde tais *scripts* de múltiplas origens têm acesso a toda informação contida nas páginas e que podem interferir uns com os outros. Desta característica surgiram alguns dos problemas que motivam a existência deste trabalho.

Um dos problemas, conhecido como *cross-site scripting* (XSS), caracteriza-se pelo redirecionamento de informação sigilosa para *sites* não confiáveis. *Sites* como MySpace.com e Twitter já foram alvo desse tipo de ataque (IBM, 2017), assim como o comércio eletrônico E-Bay (VANUNU, 2016). Contra XSS não existe uma forma universal de prevenção: cada aplicação web deve se precaver para evitar a ativação de *scripts* maliciosamente incorporados ao seu conteúdo. Mas ainda que as devidas precauções sejam tomadas, nada impedirá que o navegador carregue *scripts* adulterados sem conhecimento dos autores de uma página, como ocorreu com o *bureau* de crédito norte-americano Equifax (SEGURA, 2017) e na invasão e adulteração de *scripts* da rede de distribuição de conteúdo BootstrapCDN (DORFMAN, 2013). Em ambos os casos, os servidores de hospedagem de alguns *scripts* foram invadidos e passaram a publicar conteúdo ativo impróprio. Mesmo as extensões do navegador proporcionam outros meios de ataques, rotineiramente explorados (FORREST, 2017).

Assim, o desenvolvimento de uma aplicação segura para a web demanda esfor-

ços para que seja evitada a exposição e a manipulação indevidas das informações do usuário. Para esse propósito, o desenvolvedor conta com um conjunto de práticas e recomendações estabelecidas, efetivamente protegendo a aplicação e seus usuários de uma série de vulnerabilidades. Pelo lado dos desenvolvedores de navegadores, o estabelecimento de conjuntos de regras como a SOP (*same-origin policy*), e de protocolos como o CORS (*cross-origin resource sharing*) elevam a capacidade do navegador em manter um ambiente de execução seguro.

Contudo, tal ambiente é protegido dentro da condição de que todo conteúdo ativo carregado por uma aplicação está sob o conhecimento e confiança de seu desenvolvedor. Centralmente, o DOM, estrutura volátil em que persistem informações dos usuários, permanece exposto a *scripts* mal-intencionados ou mal-escritos, executados em contexto da página ou como extensões do navegador (HEULE et al., 2015a). Criar um *script* destinado a ler o conteúdo potencialmente sigiloso no DOM e revelá-lo a terceiros não autorizados é uma tarefa que exige pouca habilidade e que pode passar despercebida pelo aparato de segurança disponível, incluindo as restrições de SOP e CORS.

Uma das formas propostas para a solução dessas inconsistências seria a introdução de um modelo de segurança que permitisse o monitoramento do fluxo da informação na linguagem Javascript (HEULE et al., 2015b, p.3). Dessa forma, toda cópia, referência ou manipulação de dados na linguagem dependeria de uma validação dos contextos de segurança atribuídos ao dado e aos recipientes envolvidos. Isso impediria, por exemplo, que informações sensíveis armazenadas em um nó do DOM fossem lidas e transmitidas para endereços da web desautorizados – um impedimento que não seria obedecido de forma discricionária, imperativa, mas de modo mandatório, declarativo e integrado ao *runtime* da linguagem. Essa abordagem, conhecida pela sigla IFC (*information flow control*), não é compatível com os navegadores existentes. Incipientemente, mecanismos de suporte ao IFC são implementados como navegadores experimentais (HEDIN et al., 2014; BICHHAWAT et al., 2014), o que impede sua adoção maciça.

Assim, se os meios existentes para proteção contra o vazamento de informação contida no DOM são insuficientes, e se um mecanismo robusto como o IFC ainda não

foi incorporado aos navegadores tradicionais, pode ser importante propor uma abordagem que desse ao desenvolvedor um mecanismo que, embora discricionário, implementasse uma barreira que impedisse o acesso não prescrito aos nós, ou regiões, do DOM, a critério do desenvolvedor. Isso tornou-se uma possibilidade com a introdução de APIs como a de suporte ao *Shadow DOM* (W3C, 2017b), que fornecem um grau de invisibilidade a determinados conteúdos de HTML. Propor uma melhoria da segurança da informação baseada em APIs padronizadas, imediatamente disponíveis aos desenvolvedores e usuários, é a proposta deste trabalho.

1.2 Objetivo

O objetivo principal deste trabalho é propor uma abordagem para a ocultação da informação contida no DOM usando recursos padronizados de HTML e Javascript. A proposta deverá permitir ao desenvolvedor a delimitação de regiões do DOM cujo conteúdo seja opaco para *scripts* incorporados e extensões do navegador. Os requisitos da proposta serão obtidos da literatura sobre segurança da informação em Javascript.

O objetivo secundário deste trabalho é a implementação de uma prova de conceito que permita avaliar a proposta em função de seus requisitos, em testes de segurança usuais.

1.3 Contribuições

Este trabalho contribui para o estado da arte pela proposição de uma abordagem discricionária para o controle do acesso à informação armazenada no DOM. Esta é uma trilha divergente de trabalhos anteriores que exploram o controle de acesso mandatório como modelo, baseando-se, sobretudo, no controle do fluxo da informação.

Ainda que, em teoria, um modelo mandatório de controle tenda a exercer um nível de proteção maior que aquele oferecido por modelos discricionários (CNSS, 2015, p. 46) (FOSTER et al., 1998, p. 4), os navegadores da web comuns não o implementam, exceção feita a determinados acessos à rede que aconteçam sob influência da SOP (cit?). Para tanto, precisam ser estendidos por software experimental como COWL (STEFAN et al., 2014), JSFlow (HEDIN et al., 2016), o instrumentador de *bytecode*

de (BICHHAWAT et al., 2014), ou JCShadow (PATIL et al., 2011). Não são trabalhos de alcance generalizado, e até o presente momento apenas COWL é candidato a transformar-se em recomendação pelo comitê W3C (W3C, 2017a).

Ao contrário das propostas experimentais, este trabalho contribui com uma abordagem baseada em APIs já disponíveis em navegadores de ampla utilização, representando uma opção tangível para o desenvolvedor de aplicações web e seus usuários.

1.4 Método de trabalho

Os objetivos deste trabalho serão o produto de uma sequência de atividades dedicadas à exploração do problema e elaboração da solução. O método de trabalho, então, é composto das atividades enumeradas a seguir.

a) **Levantamento bibliográfico.** Esta atividade realiza-se pela pesquisa de trabalhos relacionados à segurança da informação no software navegador, incluindo contribuições relevantes que deram embasamento a esses trabalhos.

b) **Levantamento de situações de risco envolvendo vazamento de informação.** Nesta atividade, os problemas-alvo motivadores deste trabalho serão materializados em simulações e casos de teste. As evidências deverão provar que é possível efetuar as seguintes ações sem o conhecimento ou consentimento do usuário:

   – *scripts* provenientes de domínios diferentes podem observar o conteúdo de páginas da web, incluindo identificações, senhas, códigos de cartão de crédito e números de telefone, desde que essas informações estejam presentes no DOM;
   – *scripts* agindo em extensões do navegador podem observar o conteúdo de páginas da web e de seus *iframes*;
   – *scripts* de qualquer natureza podem registrar o comportamento do usuário ao interagir com a página, capturando eventos de teclado e de mouse;
   – *scripts* de qualquer natureza conseguem interceptar APIs e com isso extrair informações que transitem pelas interfaces de programação do DOM e da linguagem Javascript.

c) **Análise de requisitos.** O levantamento bibliográfico efetuado fornecerá o insumo para a enumeração dos requisitos funcionais e não-funcionais relevantes para as situações de risco levantadas.

d) **Proposição.** O objetivo desta atividade é elaborar um método para que os objetivos do trabalho sejam alcançados, em aderência aos requisitos funcionais e não-funcionais estabelecidos.

e) **Implementação.** Esta atividade tem a finalidade de produzir um componente de HTML e Javascript compatível com os requisitos estabelecidos pela proposta.

f) **Avaliação do método.** Nesta tarefa, o componente implementado será submetido à avaliação de sua eficácia frente às vulnerabilidades, e de sua compatibilidade em relação aos requisitos do método.

g) **Síntese dos resultados.** A partir das observações produzidas na atividade de avaliação, será elaborada uma síntese dos resultados alcançados em contraste com os objetivos desta proposta.

1.5 Organização do trabalho

A seção 2, Estado da Arte, enquadra o tema sob três pontos de vista: (1) das vulnerabilidades derivadas da tecnologia atual, (2) dos recursos implementados pelos navegadores para a detenção de determinados ataques à segurança da informação, e (3) das propostas experimentais para a mitigação de vulnerabilidades. O panorama formado por esses três pontos de vista corresponde ao contexto em que as contribuições deste trabalho estão inseridas.

A seção 3, Proposta, descreve um método para o desenvolvimento de componentes de HTML que mantenham invisíveis, para o restante da página, as informações mantidas ou geradas por esses componentes, ao mesmo tempo em que expõe uma interface de programação baseada em controle do acesso à informação encapsulada. São apresentadas nesta seção a disponibilidade dos recursos necessários para a implementação do método, bem como suas limitações de uso.

Na seção 4, Avaliação, são propostos critérios para a verificação da eficácia do método proposto: disponibilidade nas plataformas de navegação, limites de proteção

versus vulnerabilidades mitigadas, e requisitos de funcionamento. A seção se completa com a aplicação desses critérios sobre o método proposto, em comparação com trabalhos embasados pela abordagem de IFC – o controle de fluxo de informação define, no âmbito do problema, maior granularidade na segurança da informação em Javascript, ao custo da compatibilidade com a base instalada de navegadores.

O conteúdo da seção 5, Conclusões, deriva da reflexão crítica sobre a implementação do método proposto em contraponto aos resultados observados na avaliação qualitativa. Recomendações sobre a aplicação do método, além de oportunidades a serem exploradas por trabalhos futuros, fecham a conclusão dos esforços deste trabalho.

2 FUNDAMENTOS

2.1 Principais conceitos

Esta dissertação explora o tema da segurança da informação no navegador sob sete aspectos: **1)** dos conceitos básicos de segurança da informação no contexto deste trabalho, **2)** dos fluxos de informação no navegador, **3)** das vulnerabilidades associadas a esses fluxos, caracterizadoras do problema em questão, **4)** do aparato de segurança da informação implementado pelos navegadores e suas deficiências, **5)** dos requisitos para a avaliação da segurança da informação no navegador, **6)** do estado a arte na mitigação de riscos e **7)** dos recursos de programação que embasam a proposta desenvolvida na seção 3 deste trabalho. A apresentação desses aspectos é fundamental para a completa compreensão da proposta deste trabalho, seus objetivos e contribuições.

2.2 Conceitos básicos

2.2.1 Segurança da informação

Segundo (ISO, 2016), segurança da informação é um processo com os objetivos de "preservação da confidencialidade, integridade e disponibilidade da informação". (FOSTER et al., 1998) elabora esses objetivos, descrevendo a confidencialidade como a condição na qual a informação só pode ser acessada pelos agentes autorizados, integridade como a capacidade de proteger a informação contra modificações não autorizadas, e a disponibilidade como a capacidade de garantir acesso à informação quando necessário; (FOSTER et al., 1998) ainda atribui mais duas características a um sistema de segurança da informação: *accountability* como a possibilidade de se atribuir um agente para cada ação ocorrida dentro do sistema, e *assurance* como o grau de confiabilidade na segurança do sistema em relação aos seus objetivos declarados.

Neste trabalho, qualquer definição de segurança da informação será restrita aos sistemas de informação relacionados com a navegação de usuários através da web: provedores de serviço (*sites*, servidores da web), protocolos de comunicação em rede

(HTTP, HTTPS, *web sockets*), navegadores (*browsers*) e os ambientes de execução de Javascript embutidos nos navegadores. Isto delimita a área de conhecimento relevante para este trabalho.

2.2.2 Modelos de controle de acesso

Enquanto a definição dos requisitos de segurança da informação estabelece seus objetivos, os modelos definem os sistemas derivados desses objetivos (GOGUEN; MESEGUER, 1982). (FOSTER et al., 1998) menciona diferentes modelos de controle de acesso, categorizados de modo amplo como modelos discricionários (DAC – *discretionary access control*) e mandatórios (MAC – *mandatory access control*). Modelos discricionários se baseiam na definição dos relacionamentos de segurança entre agentes e objetos em um sistema, como, por exemplo, a política de que um *script* – a parte *agente* – não pode iniciar conexões com domínios diferentes do seu próprio – a parte *objeto*. Modelos discricionários são os mais comumente utilizados para estabelecer mecanismos de segurança nos navegadores. O campo de atuação desses modelos é limitado aos relacionamentos de segurança estabelecidos, e portanto não podem garantir a segurança da informação quando esta ultrapassa o domínio desses relacionamentos. Isto significa, por exemplo, que dados legitimamente obtidos dentro de regras discricionárias pode ser replicado para um contexto não-seguro sem qualquer impedimento derivado do modelo de segurança.

Modelos mandatórios não atribuem explicitamente as regras de controle de acesso aos objetos e agentes de um sistema. Ao invés disso, estabelecem níveis de confidencialidade utilizados para classificar os participantes do sistema de informação, viabilizando o controle dinâmico do trânsito da informação entre os agentes. Num modelo mandatório, o nível de segurança de um dado impede que ele seja obtido ou modificado por agentes com níveis de segurança mais baixos. O controle do fluxo da informação faz dos MACs modelos mais robustos do que os DACs (FOSTER et al., 1998).

2.2.3 Controle do fluxo de informações

O controle do fluxo de informações (IFC – *information flow control*) é um mecanismo que atua, em tempo de execução, nos meios de propagação dos valores entre os es-

paços de armazenamento de um sistema computacional de modo a impedir fluxos não autorizados dos dados (DENNING, 1976). IFC é um modelo de controle de acesso do tipo mandatório e baseia-se em *classes de segurança* “altas” e “baixas”, simbolizadas pelas letras `<h>` e `<l>`, respectivamente, para indicar graus de confidencialidade das informações e dos seus espaços de armazenamento (*heap*, pilha, redes, dispositivos etc). Operações entre entidades com classes de segurança diferentes, como a cópia do valor de uma variável `<h>` (confidencial) para a variável `<l>` (pública), são automaticamente impedidas de prosseguir.

IFC distingue entre fluxos de informação explícitos e implícitos. Um fluxo explícito ocorre quando uma informação classificada como “alta” é diretamente copiada para um contexto de classificação “baixa”, como na listagem de código 2.1. Em um fluxo implícito, não é a informação em si que transita entre contextos de classificação diferente, mas sim alguma informação derivada dela através da qual seja possível fazer qualquer inferência sobre seu conteúdo. Um exemplo de fluxo implícito encontra-se la listagem 2.2. Um mecanismo que suporte IFC deve ser capaz de interromper vazamento de informação em ambos os tipos de fluxo.

```
1  var revelaH = function(h) {
2    // O valor do parâmetro <h>, tido como confidencial, é explicitamente
3    // propagado para o domínio www.evil.com:
4    makeAjaxCall('www.evil.com/tell/secret/' + h);
5  }
6
7  var h = document.getElementById('password').value;
8  revelaH(h);
```

Código 2.1 – Vazamento de dados em fluxo explícito de informação

```
1  var revelaH = function(h) {
2    // Uma pista sobre o valor do parâmetro <h>, tido como confidencial,
3    // é propagada para o domínio www.evil.com:
4    var pista = h.length;
5
6    // Embora o valor <h> não tenha sido propagado, uma característica
7    // implícita do seu conteúdo foi revelada.
8
9    makeAjaxCall('www.evil.com/tell/hint?passwordLength=' + pista);
10   // Agora, www.evil.com sabe o tamanho da senha utilizada, e pode usar essa
11   // informação para fazer uma inferência sobre a senha utilizada.
```

```
12 }
13
14 var h = document.getElementById('password').value;
15 revelaH(h);
```

Código 2.2 – Vazamento de dados em fluxo implícito de informação

2.2.4 Fluxos da informação no navegador

A capacidade que software navegador tem para buscar informação, apresentá-la ao usuário e torná-la interativa é produto da cooperação de subsistemas responsáveis pela comunicação através da rede, a interpretação de código HTML, a representação visual das páginas, o gerenciamento de *caches* e a execução de *scripts*. O quadro 2.2.4 ilustra os subsistemas componentes dos navegadores baseados no projeto Chromium (CHROMIUM, 2018a; CHROMIUM, 2018b; GARSIEL; IRISH, 2011), enfatizando a hierarquia de componentes desses subsistemas, cujas responsabilidades estão descritas na tabela 1.

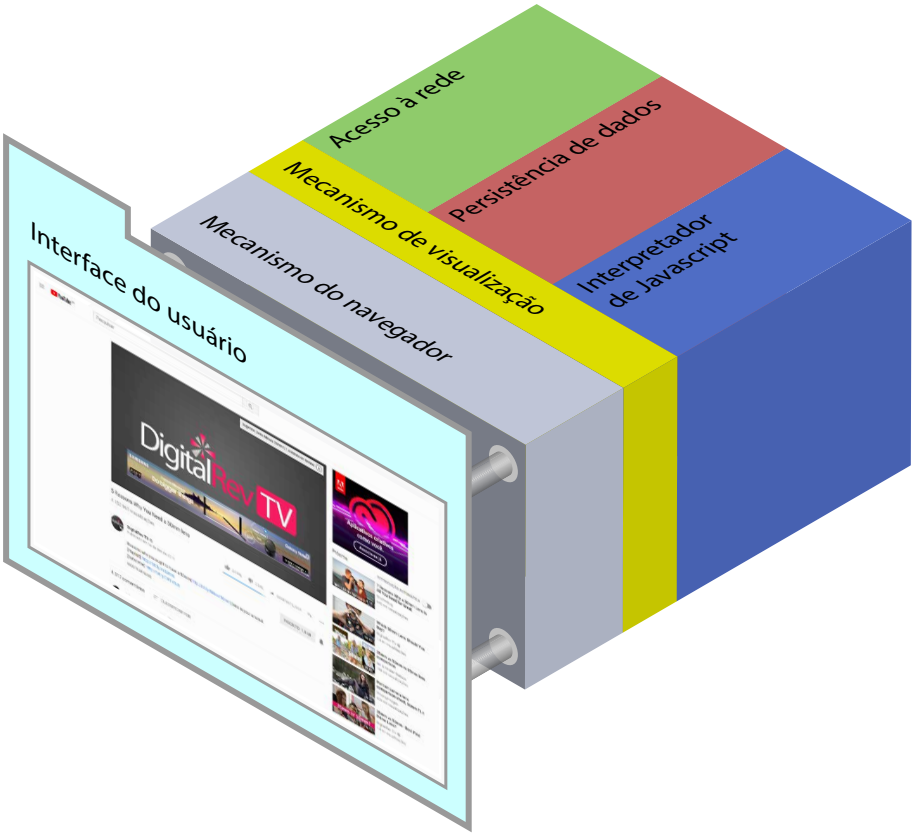

Figura 1 – Componentes do navegador de código aberto Chromium

Sob o ponto de vista da segurança da informação, cabem algumas observações:

Subsistema	Responsabilidade
Interface do usuário	Expor os elementos interativos de que o usuário dispõe para comandar o navegador, como a barra de endereços, os comandos de navegação (“voltar”, “avançar”, “recarregar”) e os comandos de menu do navegador, incluindo os menus de contexto, atalhos, acesso às configurações e às extensões. A janela de interface do usuário hospeda, ainda, o componente de superfície (*canvas*) responsável pela exibição do conteúdo tornado visível (“renderizado”) pelo subsistema Mecanismo do navegador.
Mecanismo do navegador	Comandar o mecanismo de visualização sob a demanda das ações originadas pela interface do usuário e, em resposta a esses comandos, sinalizar mudanças de estado que serão refletidas pela interface de usuário. Exemplo disso é o ciclo de carregamento de páginas da web, que dispara o *download* de recursos de rede, a renderização desses recursos e a atualização da interface de usuário à medida em que as etapas do carregamento se sucedem.
Mecanismo de visualização	“Renderizar”, ou transformar em conteúdo audiovisual, o fluxo de dados transferidos pelo subsistema de acesso à rede. O fluxo de dados carrega informação textual codificada nas linguagens HTML e CSS, e também dados binários em formato de imagem, áudio e vídeo. O mecanismo de visualização pode dar suporte à renderização de dados em formatos não vinculados à HTML, como documentos PDF e arquivos XML.
Acesso à rede	Conectar-se aos provedores de informação indicados por meio de URLs (*uniform resource locators*) e estabelecer fluxos de dados sob diferentes protocolos como HTTP, HTTPS, Web Sockets, WebRTC e FTP.
Persistência de dados	Prover funcionalidades para armazenamento de dados persistentes, incluindo *caches*, *cookies*, suporte ao sistema de arquivos e as APIs para bancos de dados de escala reduzida como *localStorage* e IndexedDB.
Interpretador de Javascript	Compilar e executar código na linguagem Javascript. “V8”, “Chakra” e “SpiderMonkey” são os nomes dos subsistemas de interpretação de Javascript empregados pelos navegadores Chrome, Microsoft Edge e Firefox, respectivamente. Os diversos interpretadores aderem às especificações estabelecidas pelo consórcio ECMA sob o padrão “ECMAScript”, que define a sintaxe e os recursos que devem ser suportados pela linguagem.

Tabela 1 – Responsabilidades dos subsistemas do navegador

a) Os subsistemas do navegador podem apresentar conjuntos próprios de vulnerabilidades, inerentes ao funcionamento e à implementação de cada um;

b) O inter-relacionamento entre esses subsistemas estabelece fluxos de informação que, por sua vez, também podem apresentar vulnerabilidades;

c) Ao longo do tempo, a descoberta de vulnerabilidades e a evolução dos navegadores contribuíram para o estabelecimento de políticas para a segurança na web;

d) A manutenção de certas funcionalidades, necessárias para o funcionamento das aplicações web modernas, comprometeu a capacidade do navegador de se manter imune a todas as vulnerabilidades de vazamento de informação possíveis.

2.2.4.1 Ciclo de vida de uma página da web

O ciclo de vida de uma página da web é a sequência de atividades executadas pelo navegador durante os processos de carga, exibição e interação do usuário com o conteúdo visualizado, perdurando até a ação de descarregamento da página, desencadeada pela navegação do usuário ou pelo encerramento do programa navegador. A informação mantida ou criada dentro do ciclo de vida de uma página tende a ser volátil, sendo descartada no descarregamento a menos que seja transmitida para outro *host* ou armazenada localmente em estruturas como *localStorage* e IndexedDB.

TO DO... Pontos a considerar:

a) O navegador da web é um cliente da internet.

b) O ciclo de vida de uma página é ”curto”(cada página, uma aplicação diferente? um contexto de execução diferente?)

c) Onde está a informação no navegador, e qual sua duração possível?

- Endereço (URL) corrente
- Requisição (pode ser da página como de ”sub-recursos”como imagens, iframes, scripts, estilos, xhr...)
- Resposta
- Estrutura da página (DOM)
- Eventos do navegador/DOM
- Closures
- Pilha de chamadas
- Extensões do navegador
- Caches
- Cookies
- Local DB
- Em HTTPS, temos mais fluxos?

Então será possível indicar quais dos fluxos estão protegidos contra vazamento de informação, e como estão. E consequentemente quais fluxos não estão cobertos.

2.2.5 Caracterização do problema

TO DO... Aqui é preciso apresentar como este problema é caracterizado, nas referências cabíveis. Tudo indica que se trata de um ”DOM-based XSS”, que eu livremente traduzo como ”XSS baseado em DOM”(péssimo). Imagino colocar a definição do OWASP e indicar as classificações do CWE (Common Weaknesses Enumeration) que orbitam em torno do problema.

(OWASP, 2015) define um ataque denominado “XSS baseado em DOM” como a modificação deliberada do modelo de objetos do navegador com o objetivo de influenciar seu ambiente de execução, sem que a visualização da página seja prejudicada ou que o navegador se comporte inesperadamente. Por essa característica, o usuário não percebe que pode haver um ataque em andamento.

2.2.6 Vulnerabilidades

Violações de privacidade são possíveis nos navegadores por causa da natureza dinâmica da linguagem Javascript e de sua ausência de restrições de segurança em tempo de execução (JANG et al., 2010). Seus usuários estão expostos a ataques sutis com objetivos diversos como roubar *cookies* e *tokens* de autorização, redirecionar o navegador para sites falsos (*phishing*), observar o histórico de navegação e rastrear o comportamento do usuário através dos movimentos do ponteiro do mouse e eventos de teclado. Para que *scripts* mal-intencionados sejam incorporados a páginas benignas, *hackers* fazem uso de vulnerabilidades como *cross-site scripting (XSS)* (OWASP, 2016) e comprometimento de extensões (HEULE et al., 2015a) do navegador.

2.2.6.1 Compartilhamento do ambiente de execução

Código *inline* ou *scripts* baixados pelas páginas da web são executados com os mesmos privilégios e mesmo nível de acesso à estrutura de documento do navegador, o chamado DOM (DeRyck et al., 2012, p. 2-3), não importando o domínio de origem dos *scripts*. Uma demonstração do problema pode ser exemplificada na figura 2 e listagem de código 2.3. Nesse exemplo, um *script* tido como benigno é incorporado a

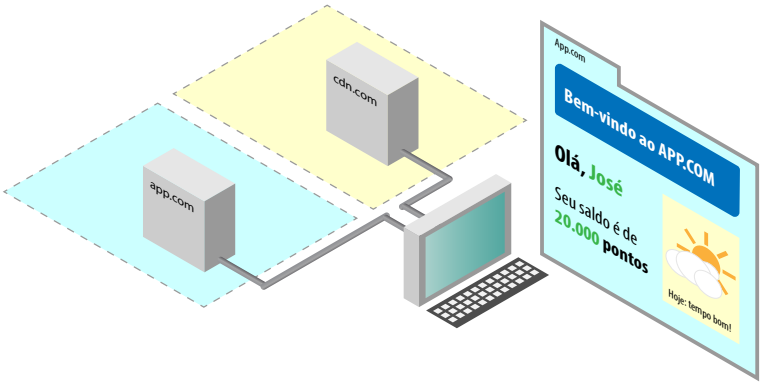

Figura 2 – Aplicação web composta por conteúdo proveniente de duas origens.

```
1  <html>
2    <head>
3      <title>App.com</title>
4      <script src="//cdn.com/previsao-do-tempo.js"></script>
5    </head>
6
7    <body>
8      <h1>Bem-vindo ao APP.COM</h1>
9
10     <div>
11       <div>
12         <p>Olá, <span class="userInfo">José</span></p>
13
14         <p>Seu saldo é de <span class="userInfo">20.000</span> pontos</p>
15       </div>
16
17       <div id="appletPrevisaoTempo"></div>
18     </div>
19   </body>
20 </html>
```

Código 2.3 – Incorporação de *script* de outra origem (linha 4)

uma página web a partir de um domínio de CDN (*content delivery network*), diferente daquele da aplicação que efetivamente publica a página. O servidor desse *script*, pelo protocolo CORS, sinaliza ao navegador que o domínio da página é confiável. O *script* externo pode, então, iniciar requisições ao seu domínio de origem – uma consequência desejada pelos autores da página, pois o *script* depende desse acesso para efetuar suas funções.

Em momento posterior, o *script* servido pelos servidores da CDN é substituído por código malicioso que, além de efetuar as funções do *script* benigno, captura o conteúdo da página armazenado no DOM (figura 3). O *script* pode buscar informações específicas e potencialmente sensíveis como identificação do usuário, senhas e endereços. Por causa da autorização concedida pelo protocolo CORS, o código mal intencionado tem a chance de transmitir o conteúdo capturado para um serviço anômalo hospedado

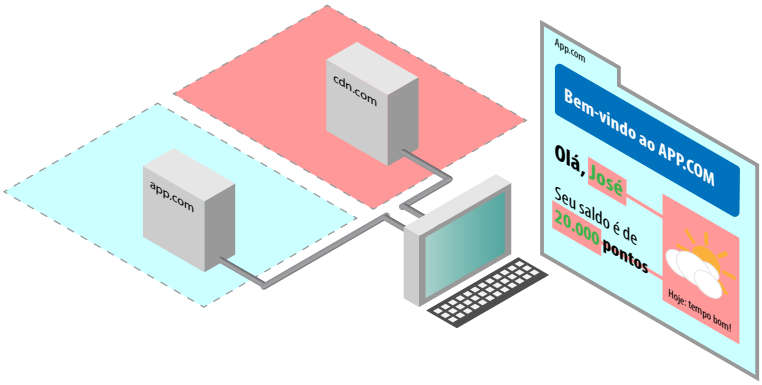

Figura 3 – Domínio de CDN comprometido, capturando informações do usuário.

no mesmo domínio do *script*.

Acessar, capturar e modificar informações contidas no DOM também são efeitos de extensões do navegador. Mas, diferentemente dos *scripts* incorporados em páginas, extensões são executadas em modo privilegiado e podem afetar todas as páginas carregadas pelo navegador, não sendo confinadas a domínios específicos. Extensões como as do Google Chrome são publicadas exclusivamente em site específico e protegido, mas não é impossível que o código fonte de extensões seja descaracterizado e publicado pela ação de *hackers* (SPRING, 2017), afetando a todos os usuários que atualizarem a extensão – um processo automático por padrão (GOOGLE, 2017).

2.2.6.2 Cross-Site Scripting (XSS)

Em Javascript, todos os recursos de código carregados dentro de uma mesma página possuem os mesmos privilégios de execução. Ataques do tipo *cross-site scripting* tiram proveito dessa característica para injetar código malicioso em contextos onde seja possível observar e retransmitir informação sigilosa como *cookies* do usuário, endereço do navegador, conteúdo de formulários, ou qualquer outra informação mantida pelo DOM.

O emprego de medidas para prevenção de ataques XSS (WILLIAMS et al., 2016) não elimina riscos inerentes à tecnologia do navegador. Uma vez que componentes incorporados, como anúncios e *players* de mídia, conseguem carregar *scripts* tidos como confiáveis dinamicamente, um único trecho de código comprometido pode colocar informações em risco sem qualquer interferência dos dispositivos de segurança.

2.2.6.3 Comprometimento de extensões

Os mecanismos de extensibilidade oferecidos pelos navegadores melhoram a funcionalidade da web para os usuários, e o código de que são feitos é executado com privilégios mais elevados do que o dos *scripts* incorporados pelos *sites*. Por isso, os usuários precisam confirmar ao navegador que aceitam que uma extensão seja instalada, sendo informados a respeito dos privilégios que a extensão pretende utilizar. O fato de que esse processo precise se repetir a cada vez que uma extensão requisitar de um conjunto de privilégios diferente faz com que os desenvolvedores optem por solicitar, de antemão, uma gama de privilégios maior que a estritamente necessária (HEULE et al., 2015a).

Uma extensão que tiver sido comprometida (por exemplo, ao usar *scripts* de terceiros que, por sua vez, tenham sido redirecionados ou adulterados) terá assim poder para ler e transmitir todo o conteúdo carregado e exibido pelo navegador, com o potencial de causar os mesmos efeitos observados em um ataque XSS, mas em escopo e poder aumentados, já que poderiam afetar todas as páginas abertas e todas as APIs publicadas pelo navegador.

To be continued...

2.2.7 Aparato de segurança da informação implementado pelos navegadores

2.2.8 Requisitos para avaliação da segurança da informação no navegador

TO DO... Trazer, da bibliografia, requisitos para avaliação da segurança. Importante estabelecer os critérios para a seleção desses requisitos. Esses requisitos são necessários para o corpo principal do trabalho, mais adiante. Esta seção será enorme.

To be continued...

2.2.9 Abordagens para mitigação de riscos

2.2.9.1 Políticas padronizadas

2.2.9.1.1 SOP – Same Origin Policy

A política de segurança SOP foi estabelecida para que os navegadores conseguissem dar suporte a páginas com conteúdo proveniente de domínios mistos com um mínimo de segurança contra o vazamento de informação entre esses domínios (HILL, 2016). Através desta política, os navegadores podem impedir um conjunto de ataques conhecido como *cross-site resource forgery*, em que um domínio tenta instruir o navegador a fazer requisições para outro domínio em nome do usuário (OWASP, 2017).

O termo *origem* é intercambiável com a palavra *domínio* e ambos representam, para fins desta política, componentes do endereço de URL associado com cada recurso da web – a saber, o *protocolo*, o *nome do host* e a *porta TCP* de onde o recurso foi transferido (BARTH, 2011). Os exemplos a seguir representam recursos de mesma origem:

Endereço	Protocolo	Nome do *host*	Porta
`http://exemplo.com/`	http	exemplo.com	80
`http://exemplo.com:80/`	http	exemplo.com	80
`http://exemplo.com/path/file`	http	exemplo.com	80

Os endereços a seguir representam recursos de origens diferentes:

Endereço	Protocolo	Nome do *host*	Porta
`http://exemplo.com/`	http	exemplo.com	80
`http://exemplo.com:8080/`	http	exemplo.com	8080
`http://www.exemplo.com/`	http	www.exemplo.com	80
`https://exemplo.com:80/`	https	exemplo.com	80
`https://exemplo.com/`	https	exemplo.com	443
`http://exemplo.org/`	http	exemplo.org	80

Segundo a SOP, as atividades derivadas da inclusão de recursos de origens mistas são categorizadas em três ações (RUDERMAN, 2017):

a) **Escrita:** atividades deste tipo instruem o navegador para que ocorra alguma forma de navegação entre páginas, o que inclui a interação com *links*, redirecionamento e submissão de formulários. Em geral SOP não restringe este tipo de ação;

b) **Incorporação:** SOP permite que recursos incorporados à página tenham origens mistas. Isto significa que é possível a inclusão de imagens, vídeos, *scripts* e do elemento `<iframe>`, entre outros, provenientes de origens mistas e dentro de uma mesma página.

c) **Leitura:** atividades de leitura permitiriam que o conteúdo dos recursos carregados pudesse ser consultado entre origens. SOP permite que um subconjunto de funcionalidades de leitura possam ocorrer entre domínios diferentes.

Um aspecto importante da SOP é o tratamento dado a *scripts* incorporados. Quando uma página inclui um *script* proveniente de outras origens, por exemplo pelo uso de uma CDN (*content distribution network*), esses *scripts* são executados em contexto da origem do documento em que eles foram incorporados. Isto permite, por exemplo, que *frameworks* populares como jQuery e Angular.js possam ser disponibilizados em CDNs sem perder funcionalidades importantes, como a capacidade de iniciar chamadas assíncronas pela técnica AJAX. Esta concessão da SOP, porém, abre a possibilidade de que esses *scripts*, se adulterados, executem atividades maliciosas sem impedimentos.

2.2.9.1.2 CSP – Content Security Policy

CSP foi criada como um complemento à SOP, elevando a capacidade do navegador de servir como plataforma razoavelmente segura para composição de aplicações *mashup* ao estabelecer um protocolo para o compartilhamento de dados entre os componentes da página que residam em domínios diferentes. CSP define um conjunto de diretivas (codificadas como cabeçalhos HTTP) para a definição de *whitelists* – o conjunto de origens confiáveis – pelas quais navegador e provedores de conteúdo estabelecem o controle de acesso e o uso permitido de recursos embutidos como *scripts*, folhas de estilos, imagens e vídeos, entre outros. Através desse protocolo, ataques de XSS que podem ser neutralizados desde que todos os componentes na página sejam aderentes à mesma política de CSP.

2.2.9.1.3 CORS – Cross-Origin Resource Sharing

Assim como a CSP, o mecanismo CORS (W3C, 2014) complementa a SOP estabelecendo um conjunto de diretivas (cabeçalhos HTTP) para a negociação de acesso via Ajax/XHR a recursos hospedados em domínios diferentes. CORS determina que exista um vínculo de confiança entre navegadores e provedores de conteúdo, dificultando vazamento de informação ao mesmo tempo em que flexibiliza as funcionalidades das APIs. O uso de CORS permite que os autores de componentes e desenvolvedores de aplicações *mashup* determinem o grau de exposição que cada conteúdo pode ter em relação aos outros conteúdos incorporados.

CSP e CORS são recomendações do comitê W3C (BARTH et al., 2016) (W3C, 2014), sendo incorporados por todos os navegadores relevantes desde 2016 (DEVERIA, 2016a) (DEVERIA, 2016b).

2.2.9.1.4 SRI – Subresource Integrity

2.2.9.2 Abordagens proprietárias

TO DO... Lembra de alguma? Preciso recordar isto...

2.2.9.2.1 Abordagens experimentais

2.2.9.2.2 JSFLow

2.2.9.2.3 SessionGuard

2.2.9.2.4 Abordagens em processo de padronização

2.2.9.2.5 COWL

2.2.9.2.6 CSP Nível 3

2.3 Shadow DOM

A evolução do navegador como *host* de aplicações web baseadas em conteúdo misto põe em evidência algumas deficiências relacionadas ao nível de isolamento necessário para que componentes de diferentes origens possam coexistir sem interferirem entre si no que diz respeito ao comportamento e à apresentação esperada, uma vez que a página web é um meio permissivo e sensível a variações sutis como a ordem de carregamento de *scripts* e folhas de estilo. As listagens **??** e **??** exploram essas fragilidades. Mitigar esses problemas significa que o desenvolvedor precisa ter pleno conhecimento dos efeitos produzidos pelos componentes que ele incorpora a uma página, para então gerenciar esses efeitos de modo que eles produzam a menor quantidade possível de interferências.

A proposta do mecanismo de Shadow DOM é implementar uma forma padronizada para que o próprio navegador minimize as interferências entre componentes que, de outra forma, iriam causar colisões de nome dos estilos e *scripts*. A utilização de Shadow DOMs faz com que o modelo de objetos em uma página isole determinadas regiões do DOM em *shadow roots* completamente independentes. Assim, interferências como as observadas em **??** e **??** deixam de existir, como demonstrado nas listagens **??** e **??**. Shadow DOM, proposta em xx/201x, é implementada pelos navegadores Chrome, Opera e Safari, com suporte planejado para os navegadores Firefox e Microsoft Edge para os próximos meses.

2.4 Estado da arte

Esta seção destaca os trabalhos que exerceram maior influência na concepção deste trabalho e na delimitação da proposta.

2.4.0.1 Security of web mashups: A survey (DeRyck et al., 2012)

O artigo é motivado pelos requisitos de segurança de aplicações web que agregam conteúdo ativo de origens distintas (*web mashups*). Os autores definem um conjunto de categorias de requisitos não-funcionais de segurança e avaliam a conformidade desses requisitos versus funcionalidades do navegador. O critério de classificação estabelecido posiciona as diversas abordagens em quatro graduações que vão desde a separação total de componentes até sua integração completa.

Contribuição. O artigo contribui com a enumeração de requisitos que uma solução voltada à segurança da informação deve atender. Algumas das tecnologias mencionadas podem ter se tornado obsoletas ou de alcance limitado desde que o artigo foi escrito, o que não invalida o resultado pretendido pelos autores, que é considerado "estado da arte" (HEDIN et al., 2014) em pesquisa sobre segurança de aplicações de composição baseadas em Javascript.

2.4.0.2 Toward Principled Browser Security (YANG et al., 2013)

Os autores analisam os mecanismos tradicionais SOP, CORS e CSP para avaliar suas heurísticas e políticas de segurança que, em troca de flexibilidade para o desenvolvedor de aplicações web, abrem diversas janelas para o vazamento de dados. Partindo dessa condição, os autores propõem um modelo baseado em controle do fluxo da informação capaz de suportar todas as políticas associadas a esses mecanismos, sem apresentar as mesmas vulnerabilidades.

Contribuição. O artigo contribui pela reinterpretação dos mecanismos de segurança tradicionais à luz do IFC, revelando algumas das suas inconsistências e concessões em prol de funcionalidades que podem ser exploradas em ataques. Esse conhecimento fornece insumos para a definição de cenários de testes a serem desenvolvidos

neste trabalho.

2.4.0.3 JSFlow: Tracking information flow in JavaScript and its APIs (HEDIN et al., 2014)

O trabalho, uma continuação de outro de mesma autoria (HEDIN; SABELFELD, 2012), é composto de duas partes: primeiro, os autores descrevem o panorama geral do corpo de conhecimento em segurança da informação no software navegador, detalhando as vulnerabilidades mais comuns; e em segundo, apresentam o projeto JSFlow, que adiciona a capacidade de IFC ao navegador. O trabalho é concluído com um teste da eficácia do projeto.

Contribuição. O projeto JSFlow demonstra alguns dos desafios de uma implementação de IFC no navegador, especialmente a possibilidade de ocorrência dos "falsos positivos", ocasiões em que acessos legítimos são impedidos pelo sistema. Em tal situação, o desenvolvedor estaria diante de um impasse e talvez preferisse adotar uma abordagem discricionária, como é a proposta deste trabalho.

2.4.0.4 Protecting Users by Confining JavaScript with COWL (STEFAN et al., 2014)

Em concordância com o artigo associado (YANG et al., 2013), os autores argumentam que, face às dificuldades que os desenvolvedores encontram para aderir aos mecanismos tradicionais SOP, CSP e CORS, acaba-se optando pela funcionalidade em detrimento da segurança. Isto se manifesta em extensões de navegador solicitando mais permissões do que o necessário, em *web mashups* que requerem autorizações desnecessárias para o usuário, e em notificações de segurança tão constantes que se tornam efetivamente invisíveis. Entendendo que o estado-da-arte da análise do fluxo de informações em navegador é deficiente – seja porque as ferramentas são incompletas ou porque degradam desempenho –, os autores apresentam o projeto COWL, um navegador construído sobre o software Firefox que implementa, experimentalmente, o controle do fluxo da informação.

Contribuição. O navegador COWL será utilizado como participante de cenários de teste a serem explorados neste trabalho, para que seja avaliada a efetividade de uma

implementação de IFC como meio de mitigar as vulnerabilidades identificadas.

2.4.0.5 Information Flow Control for Event Handling and the DOM in Web Browsers (RAJANI et al., 2015)

O artigo explora vazamento de informação em *scripts* acionados por eventos do DOM, demonstrando que esse é um efeito de como os navegadores disparam e propagam eventos. Os autores apontam as particularidades da propagação de eventos e suas implicações para o controle do fluxo da informação, implementando, no navegador WebKit, um mecanismo de IFC imune a vazamento de informação derivado do disparo de eventos.

Contribuição. Este artigo coloca em pauta a segurança da informação no âmbito dos eventos do DOM, uma forma sutil de materializar o problema central deste trabalho. Também contribui com a definição de uma máquina de estados derivada do comportamento do navegador na ocorrência de um evento, útil para a concepção de casos de testes.

2.4.0.6 The Most Dangerous Code in the Browser (HEULE et al., 2015a)

O artigo apresenta o desafio, ainda não superado, das vulnerabilidades derivadas do modelo de confiança que os navegadores utilizam para instalar, acionar e atualizar software de extensão. Os autores demonstram como uma extensão pode comprometer a segurança da informação, propondo um modelo de controle mandatório de acesso para mitigar oportunidades de vazamento de dados.

Contribuição. Este artigo mapeia as vulnerabilidades associadas às interações entre o DOM e as extensões do navegador. Esse conhecimento é fundamental para que seja possível definir requisitos e cenários de testes que envolvam extensões.

3 PROPOSTA

3.1 API de ocultação do DOM fundamentada em Shadow DOM

O mecanismo de ocultação do DOM proposto neste trabalho faz com que a informação contida em um ramo específico da árvore do DOM, bem como a informação gerada por esse ramo do DOM, esteja fora do alcance de *scripts* que se baseiem em interações com o DOM para obter dados que venham a ser indevidamente propagados. Trata-se de uma medida discricionária de segurança da informação, caracterizável pelos aspectos X, Y e Z das políticas discricionárias de segurança da informação.

A característica de ocultação deriva de uma propriedade das APIs de *shadow DOM*, que estabelece, em sua especificação, que para todo *shadow host* será especificada a visibilidade externa de sua *shadow tree* (estrutura de DOM subjacente associada ao *host*). Essa propriedade é determinada no momento da criação do *shadow host*, conforme exemplificado na listagem de código 3.1, linhas 2 e 6.

```
1  var openShadowHost = document.createElement('div');
2  var openRoot = openShadowHost.attachShadow({mode: 'open'});
3  openRoot.innerHTML = '<p><i>Shadow root</i> aberta.';
4
5  var closedShadowHost = document.createElement('div');
6  var closedRoot = closedShadowHost.attachShadow({mode: 'closed'});
7  closedRoot.innerHTML = '<p><i>Shadow root</i> fechada.';
8
9  document.body.appendChild(openShadowHost);
10 document.body.appendChild(closedShadowHost);
11
12 // A instrução a seguir resulta em um objeto "#shadow-root" sendo exposto no
       console do navegador:
13 console.log(openShadowHost.shadowRoot);
14
15 // A instrução a seguir resulta em um valor null sendo exposto no console do
       navegador:
16 console.log(closedShadowHost.shadowRoot);
```

Código 3.1 – Determinação da visibilidade da *shadow tree* e sua consequência na tentativa de acesso ao DOM subjacente

Dessa forma, o método `Element.attachShadow({mode: open})` abre a possibilidade de que uma *shadow tree* seja inacessível a *scripts* que não detenham uma referência à *shadow root*, efetivamente provendo um meio programático para a definição de regiões protegidas na estrutura DOM da página web. No entanto, a invisibilidade do conteúdo da *shadow root* não é de qualquer modo garantida pela API de Shadow DOM. A extensibilidade da linguagem Javascript permite que *scripts* interceptem os métodos da API, ganhando acesso a todas as *shadow roots* criadas em uma página web. Essa estratégia é detalhada na seção seguinte.

3.1.1 Interceptação de métodos nativos do ambiente de execução

A linguagem Javascript incorpora o conceito de herança prototípica para permitir que programas escritos nessa linguagem façam uso de funcionalidades da programação orientada a objetos como herança e polimorfismo. Protótipos de objeto são semelhantes à ideia de “classes” em linguagens estáticas com suporte à POO, porém, ao contrário delas, Javascript permite a redefinição dinâmica dos membros dos protótipos/classes. A listagem 3.2 exemplifica esse mecanismo.

```
1  var myClass = {
2    nomeAtribuido: '',
3    idade: 0,
4    exibirNome: function() {
5      console.log('Meu nome é ' + this.nomeAtribuido);
6    }
7  };
8
9  // A função Object.create() cria um novo objeto
10 // baseado em um protótipo, como na linha a seguir:
11 var objetoHerdeiro = Object.create(myClass);
12 objetoHerdeiro.nomeAtribuido = 'Teste';
13
14 // objetoHerdeiro herda as funções definidas em myClass e redefine o valor
15 // do membro nomeAtribuido). A função (*exibirNome(), por exemplo, faz referência
16 // a esse membro, inicializado como string vazia em myClass, mas retornado
17 // como 'Teste' na chamada de método a seguir:
18 objetoHerdeiro.exibirNome();
19
20 // O objeto myClass ganha um novo membro, a função exibirIdade:
21 myClass.exibirIdade = function() {
22   console.log('Minha idade é ' + this.idade);
```

```
23 };
24
25 // A função recém-definida no protótipo é imediatamente disponível
26 // como membro herdado:
27 objetoHerdeiro.exibirIdade();
```

Código 3.2 – Demonstração de herança prototípica em Javascript

Uma consequência da flexibilidade oferecida pela herança prototípica é que um *script* pode modificar os métodos de um protótipo de forma que o acionamento desses métodos seja encapsulado por código arbitrário. Essa capacidade de interceptação permite que métodos sejam instrumentados, monitorados ou corrompidos para quaisquer fins. O exemplo na listagem 3.3 demonstra como o método `Number.toString()` pode ser desvirtuado para retornar informação incorreta – neste caso, o método sempre retornará a mesma *string* "42".

```
1  // A variável number contém uma constante de valor numérico,
2  // instância derivada do prototype Number:
3  var number = 41;
4
5  console.log('O valor da variável number é ' + number.toString());
6
7  // O script agora interceptará Number.toString() para que o texto retornado
8  // seja diferente do esperado:
9  var metodoInterceptado = Number.prototype.toString;
10
11 Number.prototype.toString = function interceptadora() {
12   var expectedString = metodoInterceptado.call(this);
13   return '42 (originalmente, seria ' + expectedString + ')';
14 };
15
16 // A chamada seguinte, idêntica à da linha 7, terá resultado inesperado:
17 console.log('O valor da variável number é ' + number.toString());
```

Código 3.3 – Exemplo de método nativo interceptado por um script

3.2 Casos de teste de exposição aos riscos de vazamento de informação no navegador

To be continued...

3.3 Apresentação do site acadêmico “jstest.me”

To be continued...

3.4 Roteiro para a elaboração da proposta

a) **Mapeamento de cenários.** Partindo dos requisitos não funcionais estabelecidos, serão definidos cenários de testes capazes de evidenciar as condições necessárias para a não-conformidade com esses requisitos. Os cenários assim definidos delimitarão o escopo de ação da abordagem proposta por este trabalho.

b) **Especificação da proposta.** A abordagem derivada dos cenários de testes será formalmente especificada, em termos das interfaces e funcionalidades esperadas, visando orientar o desenvolvimento de uma implementação de teste e provas de conceito.

c) **Implementação do mecanismo de ocultação.** A materialização da abordagem proposta será efetuada pelo desenvolvimento dos scripts necessários para a implementação das interfaces esperadas, em conformidade com os requisitos funcionais esperados.

d) **Validação.** Os cenários de testes serão implementados na forma de provas de conceito para validação da proposta e do mecanismo implementado. O objetivo é conhecer a aderência do mecanismo em relação aos requisitos.

REFERÊNCIAS

ADOBE. **Flash & The Future of Interactive Content**. 2017. Disponível em: <https://theblog.adobe.com/adobe-flash-update/>.

BARTH, A. **The Web Origin Concept**. 2011. Disponível em: <https://tools.ietf.org/html/rfc6454>.

BARTH, A. et al. **Content Security Policy Level 2**. [S.l.], 2016. Https://www.w3.org/TR/2016/REC-CSP2-20161215/.

BICHHAWAT, A. et al. Information Flow Control in WebKit's JavaScript Bytecode. In: **Principles of Security and Trust**. [S.l.: s.n.], 2014. p. 159–178. ISBN 978-3-642-54792-8.

CHROMIUM. **Multi-process Architecture**. 2018. Disponível em: <https://www.chromium.org/developers/design-documents/multi-process-architecture>.

____. **Multi-process Resource Loading**. 2018. Disponível em: <https://www.chromium.org/developers/design-documents/multi-process-resource-loading>.

CNSS. Committee on National Security Systems Glossary. n. CNSSI 4009, 2015.

DENNING, D. E. A lattice model of secure information flow. **Commun. ACM**, v. 19, n. 5, p. 236–243, 1976. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/360051.360056>.

DeRyck, P. et al. Security of web mashups: A survey. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 7127 LNCS, p. 223–238, 2012. ISSN 03029743.

DEVERIA, A. **Can I Use: Content Security Policy 1.0**. 2016. Disponível em: <http://caniuse.com/\#search=Contentsecuritypolicy>.

____. **Can I Use: Cross-Origin Resource Sharing**. 2016. Disponível em: <http://caniuse.com/\#feat=cors>.

DORFMAN, J. **BootstrapCDN Security Post-Mortem**. 2013. Disponível em: <https://www.maxcdn.com/blog/bootstrapcdn-security-post-mortem/>.

FORREST, C. **Warning: These 8 Google Chrome extensions have been hijacked by a hacker**. 2017. Disponível em: <https://www.techrepublic.com/article/warning-these-8-google-chrome-extensions-have-been-hijacked-by-a-hacker/>.

FOSTER, I. et al. A security architecture for computational grids. In: **Proceedings of the 5th ACM conference on Computer and communications security - CCS '98**.

New York, New York, USA: ACM Press, 1998. v. 44, n. 2, p. 83–92. ISBN 1581130074. Disponível em: <http://portal.acm.org/citation.cfm?doid=288090.288111>.

GARSIEL, T.; IRISH, P. **How Browsers Work: Behind the scenes of modern web browsers**. 2011. Disponível em: <https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>.

GOGUEN, J. A.; MESEGUER, J. Security Policies and Security Models. In: **1982 IEEE Symposium on Security and Privacy**. IEEE, 1982. p. 11–11. ISBN 0-8186-0410-7. Disponível em: <http://ieeexplore.ieee.org/document/6234468/>.

GOOGLE. **Autoupdating**. 2017. Disponível em: <https://developer.chrome.com/extensions/autoupdate>.

HEDIN, D. et al. Information-flow security for JavaScript and its APIs. **Journal of Computer Security**, v. 24, n. 2, p. 181–234, 2016. ISSN 0926227X.

____. JSFlow: Tracking information flow in JavaScript and its APIs. **Proceedings of the 29th Annual ACM Symposium on Applied Computing**, p. 1663–1671, 2014.

HEDIN, D.; SABELFELD, A. Information-Flow Security for a Core of JavaScript. In: **2012 IEEE 25th Computer Security Foundations Symposium**. [S.l.]: IEEE, 2012. p. 3–18. ISBN 978-1-4673-1918-8.

HEULE, S. et al. The Most Dangerous Code in the Browser. **Usenix**, 2015.

____. IFC Inside: A General Approach to Retrofitting Languages with Dynamic Information Flow Control. **Post**, n. Section 2, 2015. Disponível em: <http://web.mit.edu/{~}ezyang/Public/IFCInside.p>.

HILL, B. **CORS for Developers**. 2016. Disponível em: <https://w3c.github.io/webappsec-cors-for-developers/>.

IBM. **Inside the Mind of a Hacker: Attacking Web Pages With Cross-Site Scripting**. 2017. Disponível em: <https://securityintelligence.com/inside-the-mind-of-a-hacker-attacking-web-pages-with-cross-site-scripting/>.

ISO. ISO/IEC 27000: 2016 Glossary. **ISO.org [Online]**, v. 2016, p. 42, 2016. Disponível em: <http://standards.iso.org/ittf/PubliclyAvailableStandards/>.

JANG, D. et al. An empirical study of privacy-violating information flows in JavaScript web applications. **Proceedings of the 17th ACM conference on Computer and communications security - CCS '10**, p. 270, 2010. ISSN 15437221.

OWASP. **DOM Based XSS**. 2015. Disponível em: <https://www.washingtonpost.com/graphics/national/security-of-the-internet/history/>.

____. **Cross-site Scripting (XSS)**. OWASP, 2016. Disponível em: <https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>.

____. **Cross-Site Request Forgery (CSRF)**. OWASP, 2017. Disponível em: <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)>.

PATIL, K. et al. Towards fine-grained access control in JavaScript contexts. **Proceedings - International Conference on Distributed Computing Systems**, p. 720–729, 2011. ISSN 1063-6927.

RAJANI, V. et al. Information Flow Control for Event Handling and the DOM in Web Browsers. **Proceedings of the Computer Security Foundations Workshop**, v. 2015-Septe, p. 366–379, 2015. ISSN 10636900.

RUDERMAN, J. **Same-origin Policy**. 2017. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin{_}pol>.

SEGURA, J. **Malvertising on Equifax, TransUnion tied to third party script**. 2017. Disponível em: <https://blog.malwarebytes.com/threat-analysis/2017/10/equifax-transunion-websites-push-fake-flash-player/>.

SPRING, T. **Copyfish Browser Extension Hijacked to Spew Spam**. 2017. Disponível em: <https://threatpost.com/copyfish-browser-extension-hijacked-to-spew-spam/127125/>.

STEFAN, D. et al. Protecting Users by Confining JavaScript with COWL. **OSDI**, 2014.

VANUNU, O. eBay Platform Exposed to Severe Vulnerability. **Check Point Threat Research**, 2016. Disponível em: <http://blog.checkpoint.com/2016/02/02/ebay-platform-exposed-to-severe-vulnerability/>.

VERGE, T. **Oracle's finally killing its terrible Java browser plugin**. 2016. Disponível em: <https://www.theverge.com/2016/1/28/10858250/oracle-java-plugin-deprecation-jdk-9>.

W3C. **Cross-Origin Resource Sharing**. 2014. Disponível em: <https://www.w3.org/TR/cors/>.

____. **Confinement with Origin Web Labels**. 2017. Disponível em: <https://w3c.github.io/webappsec-cowl/>.

____. **Shadow DOM**. 2017. Disponível em: <https://www.w3.org/TR/shadow-dom/>.

WILLIAMS, J. et al. **XSS (Cross Site Scripting) Prevention Cheat Sheet**. OWASP, 2016. Disponível em: <https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet>.

YANG, E. Z. et al. Toward principled browser security. In: **Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems**. Berkeley, CA, USA: USENIX Association, 2013. (HotOS'13), p. 1–7. Disponível em: <http://dl.acm.org/citation.cfm?id=2490483.2490500>.