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.

56 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

LISTA DE CÓDIGOS

LISTA DE QUADROS

LISTA DE TABELAS

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

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 um tipo de software conhecido como “aplicação web”. 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 denominados de *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 as aplicações web alvos para ataques à confidencialidade e à integridade da informação que circula pela rede. A Fundação OWASP, organização internacional de apoio ao desenvolvimento de aplicações confiáveis, incluiu a exposição de dados confidenciais e a ocorrência de *scripts* “cross-site” *(XSS – Cross-Site Scripring)* em sua lista das dez maiores ameaças à segurança das aplicações, edição 2017 (OWASP, 2017c). Ambos os riscos se apoiam em vulnerabilidades do navegador para se manifestarem nas aplicações web. Em resposta, a comunidade de desenvolvimento dos *browsers* incorporou ao software navegador diversos mecanismos para a defesa de seus usuários. Porém, problemas fundamentais, focos das vulnerabilidades, permanecem inerentes aos padrões de funcionamento que os navegadores precisam implementar, desta forma perpetuando riscos.

Com efeito, 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. O navegador é flexível a ponto de possibilitar que uma mesma página comporte elementos de composição publicados por desenvolvedores distintos. Isso permite que recursos construídos com Javascript,

frames, aplicativos Java e conteúdo em Flash coexistam e levem 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 precisa se basear na confiança que seus autores depositam nos recursos de terceiros incorporados às aplicações.

Aplicativos para Java e conteúdo em Flash, reiteradamente vulneráveis e em descompasso com a evolução da web, acabaram em desuso e deixaram de ter suporte nativo nos navegadores (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*. O DOM, presente nos navegadores desde a introdução da linguagem Javascript e continuamente enriquecido com funcionalidades, é a base em que se apoia a construção de qualquer aplicação web moderna.

É relevante reiterar que *scripts* podem ser carregados pelo navegador a partir de múltiplas origens – *sites* diferentes daquele em que foi publicada a página que fizer uso deles. 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 todos os *scripts*, independentemente de suas origens, têm acesso a toda informação contida nas páginas e, caso tenham sido desenvolvido sem cuidado ou com finalidades duvidosas, podem interferir no funcionamento uns dos outros. Desta característica podem nascer alguns dos problemas que motivam a existência deste trabalho.

Um dos problemas, o já citado *cross-site scripting* (XSS), caracteriza-se pelo redirecionamento de informação sigilosa para *sites* não confiáveis. Serviços 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 que *scripts* maliciosos não sejam ativados. 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 código Javascript impróprio, diferente daquele originalmente tido como confiável.

Assim, o desenvolvimento de uma aplicação segura para a web demanda esforços 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 padronizadas, 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 HTTPS e CORS (*cross-origin resource sharing*) elevam a capacidade do navegador em manter um ambiente de execução seguro. Essas tecnologias são produto de iniciativas da comunidade de desenvolvimento dos navegadores, das aplicações web e das organizações padronizadoras da inernet, como os comitês W3C e IETF.

Contudo, a manutenção de um ambiente de execução seguro em uma página web ainda se sustenta dentro da condição de que todo conteúdo ativo carregado por uma aplicação esteja 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). Apesar do aparato de segurança presente no navegador, 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.

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 manipu-

lação de dados dependeria de uma validação dos contextos de segurança atribuídos ao dado e aos participantes 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 seria obedecido não 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 a implementação da linguagem Javascript incluída nos navegadores existentes. Incipientemente, mecanismos de suporte ao IFC são implementados em 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 tenha sido incorporado aos navegadores tradicionais, pode ser importante propor uma abordagem que desse ao desenvolvedor um mecanismo, ainda que discricionário, o qual 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 inviolabilidade de acesso à informação em determinados contextos de programação de *scripts* no navegador. Aplicando esses recursos, este trabalho propõe uma melhoria da segurança da informação baseada em APIs padronizadas, imediatamente disponíveis aos desenvolvedores e usuários, sem lançar mão de tecnologias experimentais.

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.

Contribuições como “COWL” (STEFAN et al., 2014), “JSFlow” (HEDIN et al., 2016), o instrumentador de *bytecode* de (BICHHAWAT et al., 2014), “JCShadow” (PATIL et al., 2011) ou “SessionGuard” (PATIL, 2017) se baseiam em tecnologia experimental, em contexto acadêmico de utilização. Até o presente momento, apenas COWL (*Confinement with Origin Web Labels*) é candidato a transformar-se em recomendação pelo comitê W3C(W3C, 2017a).

Na contramão das propostas experimentais, e inspirado por elas, 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.

2.1.1 Conceitos básicos

2.1.1.1 Segurança da informação

O glossário do padrão ISO/IEC 27000 (ISO, 2016) define segurança da informação como um processo cujos objetivos são a “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. Esses cinco parâmetros – confidencialidade, integridade, disponibilidade, *accountability* e *assurance* – representam aspectos da qualidade de uma estratégia para a segurança da informação, e servirão como base para a avaliação das soluções sob consideração neste trabalho.

Um sistema de informação complexo como a web é produto da colaboração de múltiplos participantes, todos co-responsáveis pela segurança da informação que trafega entre eles. Porém, para a delimitação de escopo deste trabalho, qualquer definição de segurança da informação será restrita aos sistemas envolvidos nas atividades de navegação que os usuários exercem através da web. Isso inclui os seguintes participantes: provedores de serviço (*sites*, servidores da web), protocolos de comunicação em rede (HTTP, HTTPS, *web sockets*), o software navegador e o ambiente de execução de Javascript a ele incorporado. Isto delimita a área de conhecimento relevante para este trabalho, ilustrado pelo diagrama 2.1.1.1.

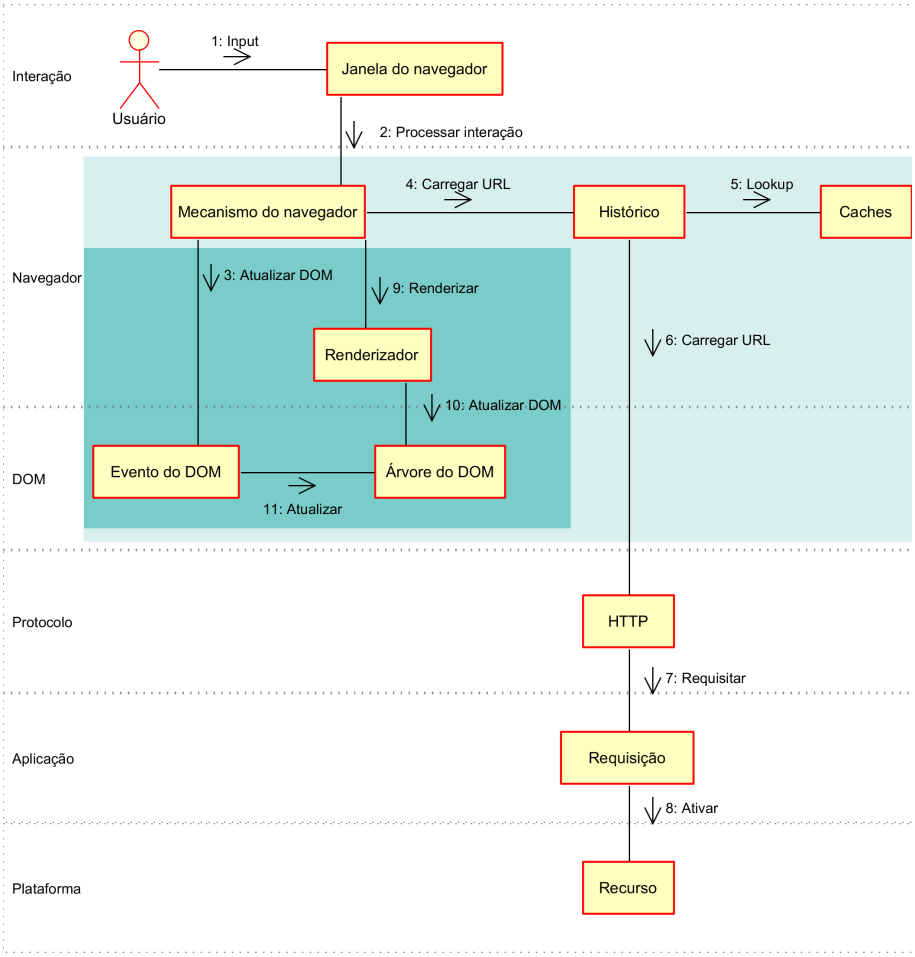

Figura 1 – Sistema de informação em contexto de aplicação web. A área sinalizada em azul corresponde ao escopo do estado da arte em segurança da informação em aplicações web, e a sub-área em azul intenso compreende o escopo da proposta descrita na seção 3 deste trabalho.

2.1.1.2 Modelos de controle de acesso

O trabalho seminal de GOGUEN; MESEGUER (1982) estabelece distinção entre as "políticas de segurança" e os "modelos de segurança" para evidenciar aspectos complementares da segurança da informação em sistemas de software. Segundo GOGUEN; MESEGUER, uma *política de segurança* define os requisitos de segurança da informação para um dado sistema, enquanto um "modelo de segurança" é a especificação da forma como esse sistema irá implementar uma política de segurança. Categorizando modelos de segurança, FOSTER et al. (1998) os divide entre "discricionários", sob a sigla DAC (DAC – *discretionary access control*), e "mandatórios", sob a sigla MAC (MAC – *mandatory access control*). Essas duas categorias nascem de abordagens opostas, com implicações fundamentais sobre a natureza das soluções de apoio à segurança da informação.

Para FOSTER et al. o conceito central dos modelos discricionários é a segurança observada no relacionamento entre agentes e objetos em um sistema. Por exemplo, quando uma política estabelece que um *script* – a parte "agente" – não pode iniciar conexões com domínios diferentes do seu próprio – a parte "objeto", um modelo DAC bem sucedido impedirá que a parte "relacionamento", representada pelas conexões de rede, possa acontecer. Modelos discricionários são os mais comumente utilizados para estabelecer mecanismos de segurança nos navegadores. Do ponto de vista da segurança da informação, modelos de DACs não podem garanti-la fora do contexto dos relacionamentos pré-estabelecidos pelas políticas. 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.

Sobre os modelos mandatórios, FOSTER et al. caracteriza-os pela existência de níveis de confidencialidade que classificam 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 objeto de informação – seja ele um arquivo, um registro de dados, um valor numérico ou qualquer outro dado – impede que ele seja obtido ou modificado por agentes aos quais tenham sido atribuídos níveis de segurança mais baixos. Modelos baseados em MAC exercem segurança em âm-

bito sistêmico, atuando sobre todas as trocas de informação, o que contrasta com a aplicação localizada das regras de segurança observadas em modelos baseados em DAC. Por causa dessa característica, FOSTER et al. sugere que o controle do fluxo da informação faz dos MACs modelos mais robustos do que os DACs.

2.1.1.3 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 espaços de armazenamento de um sistema computacional de modo a impedir fluxos não autorizados dos dados (DENNING, 1976). IFC implementa 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 acontecerem, ocasionando falha de execução.

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

2.1.2 Fluxos da informação no navegador

2.1.2.1 O navegador e seus subsistemas componentes

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*. Os navegadores divergem na forma de implementação desses subsistemas, mas em geral empregam arquiteturas que enfatizam aspectos não-funcionais como tolerância a falhas, utilização reduzida de recursos e segurança do ambiente de execução. Navegadores como Chrome, Internet Explorer e Safari utilizam separação de processos por aba de navegação (BRIGHT, 2016) para elevar o nível de isolamento entre as sessões de usuário; Firefox segue essa tendência ao adotar uma arquitetura de múltiplos processos em seus *releases* mais recentes (NGUYEN, 2017), coordenando até quatro processos de navegação compartilhados entre todas as abas. No âmbito da segurança da informação, a capacidade de execução em múltiplos processos faz com que o na-

vegador isole, em contextos de alta restrição de permissões *(sandboxes)*, quaisquer aplicações web anômalas que tentassem, por meio de *crashes* deliberados, causar instabilidade do navegador ou do sistema operacional pela execução de código em contexto privilegiado (CHROMIUM, 2018a).

Em linhas gerais, GARSIEL; IRISH descreve a colaboração entre seis subsistemas que compõem um navegador moderno, ilustrados pelo diagrama 2.1.2.1, e descritos no quadro 2.1.

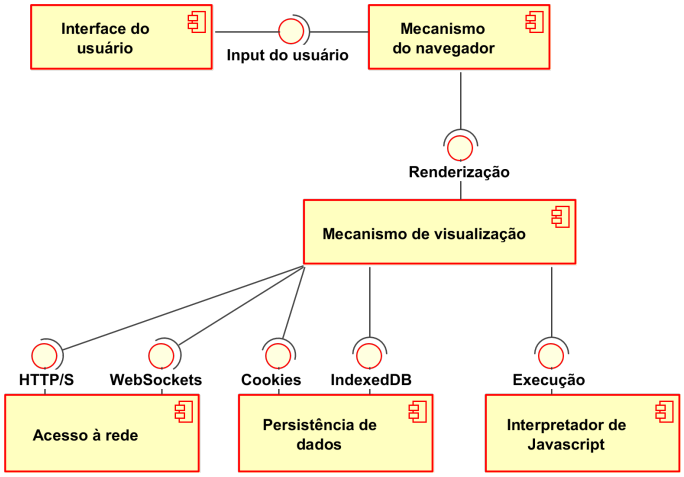

Figura 2 – Componentes do navegador de código aberto Chromium. Adaptado de (CHROMIUM, 2018a; CHROMIUM, 2018b; GARSIEL; IRISH, 2011)

2.1.2.2 Fluxos de informação violadores da privacidade do usuário

O estudo empírico de JANG et al., conduzido em 2010 tendo como alvo os 50.000 sites mais visitados do mundo[1], evidenciou uma variedade de fluxos capazes de comprometer a privacidade dos dados de usuário em nos sites *sites* de grande circulação. Os autores descrevem quatro efeitos desses fluxos: roubo de *cookies*, sequestro do endereço corrente (*location hijacking*), espionagem de histórico (*history sniffing*), e observação do comportamento do usuário. Isoladamente ou em conjunto, os fluxos violadores da privacidade podem ser acionados a partir de um único *script* comprometido, sem que o desenvolvedor do *site* afetado tenha conhecimento ou o poder de neutralizá-los preventivamente.

[1] O serviço Alexa provê serviços de análise da visitação de sites, sob consulta. <https://www.alexa.com/topsites>

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 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.

Quadro 2.1: Responsabilidades dos subsistemas componentes do navegador. Adaptado de (GARSIEL; IRISH, 2011)

a) Roubo de *cookies*. Como qualquer *script* incorporado, código malicioso tem acesso à toda informação contida na página hospedeira, incluindo informação armazenada em *cookies*, e permissão para enviar essa informação para outros agentes pelo uso de alguma das diversas maneiras de se invocar recursos remotos enumeradas na seção 2.1.3.1. *Cookies* expostos dessa maneira podem ser utilizados para forjar requisições posteriores em nome do usuário, agravando o potencial de comprometimento de informação.

b) Sequestro do endereço corrente. O endereço URL corrente do navegador é um valor exposto pelo DOM na propriedade `document.location`. Todo código Javascript tem poder de ler e modificar essa propriedade, cuja alteração causa uma imediata carga do endereço especificado. Um *script* pode aproveitar-se desse comportamento para redirecionar informação contida na página.

c) Espionagem de histórico. Embora o histórico de navegação propriamente dito, correspondente à sequência de URLs visitadas, não seja exposto pelo DOM, uma forma de ataque conhecida como XSHM *(Cross Site History Manipulation)* (OWASP, 2017a) revela informação suficiente para que seja possível deduzir, por exemplo, se o usuário fez *login* em determinada aplicação-alvo. Um *script* pode valer-se dessa dedução para emitir requisições mal-intencionadas para esse alvo em nome do usuário corrente.

d) Observação do comportamento do usuário. *Scripts* têm a capacidade de registrar código "ouvinte" de eventos derivados da interação do usuário com a página web, como movimentos de *mouse*, cliques, acionamento de teclado e rolagem de tela. Essa informação pode ser retransmitida para outros agentes na internet, que poderão acompanhar as ações do usuário em tempo real.

JANG et al. afirmam que as causas que viabilizam a existência desses fluxos são a natureza dinâmica da linguagem Javascript, sua falta de mecanismos de isolamento entre *scripts*, e a granularidade incoerente das políticas como a SOP.

2.1.3 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.

(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.

2.1.3.1 Mecanismos para propagação de informação suportados pelo navegador

BIELOVA (2013) enumera os mecanismos que o navegador oferece para permitir comunicação entre uma aplicação web e outros recursos receptores de dados. A mera possibilidade de comunicação não constitui uma ameaça às aplicações, sendo de fato o meio de se viabilizar as principais funcionalidades do navegador. No entanto, autores de *scripts* mal intencionados podem tirar proveito dos mecanismos de comunicação para propagar informação sensível.

2.1.3.1.1 APIs relevantes

Objetos do DOM *Core* Toda a árvore de elementos HTML representada na estrutura do DOM, encabeçada pelo objeto `window.document`, corresponde à API *core* do DOM. Todos os *scripts* executando em um determinado domínio têm acesso ao mesmo objeto `document`. Não havendo garantias sobre as intenções de cada *script* incorporado, cabe ao autor de uma aplicação web confiar que o comportamento desses *scripts* não é malicioso do ponto de vista da segurança da informação.

Domínio O domínio associado a uma aplicação web é refletido pela propriedade do DOM `document.domain`, que pode ser modificada por *scripts*. Um exemplo dessa operação é proporcionado por dois *scripts*, um executando em um domínio `a.app.com` e outro no domínio `b.app.com`. Ambos podem alterar seus domínios de origem para `app.com`, o que os fará compartilharem o mesmo DOM. Por meio dessa operação os *scripts* “relaxam” a amplitude do domínio relevante para as restrições derivadas da SOP.

2.1.3.2 Compartilhamento do ambiente de execução

Tanto o código *inline* quanto os *scripts* baixados pelas páginas da web são executados com os mesmos privilégios e mesmo nível de acesso à estrutura de documento do navegador (DeRyck et al., 2012, p. 2-3), não importando o domínio de origem dos *scripts*. A esse respeito, NIKIFORAKIS et al. (2012), em sua *survey* a respeito da inclusão de código Javascript hospedado em múltiplas origens, descreve esta forma de incorporação de *scripts* como uma expansão do "perímetro de segurança" da página, atribuindo a terceiros o mesmo nível de confiança dado aos *scripts* desenvolvidos pelos autores diretos de uma aplicação web.

Scripts mantidos por terceiros podem sofrer modificações não previstas pelas aplicações que os hospedam, o que tem o potencial de introduzir *bugs* ou comportamento inadequado ou hostil. Uma demonstração do problema pode ser exemplificada na figura 3 e listagem de código 2.3. Nesse exemplo, um *script* tido como benigno é incorporado a 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 4). 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 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

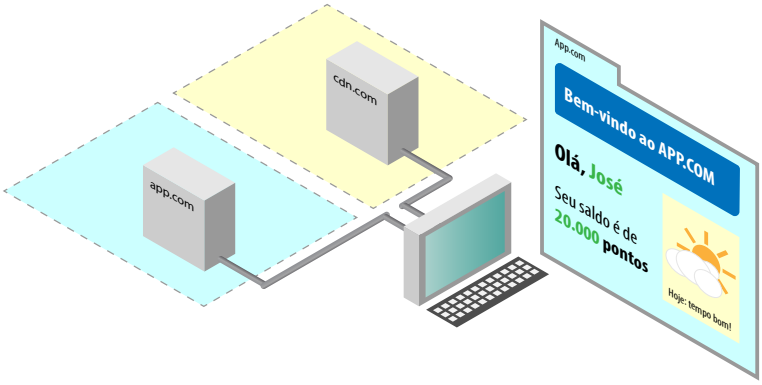

Figura 3 – Aplicação web composta por conteúdo proveniente de duas origens.

```
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)

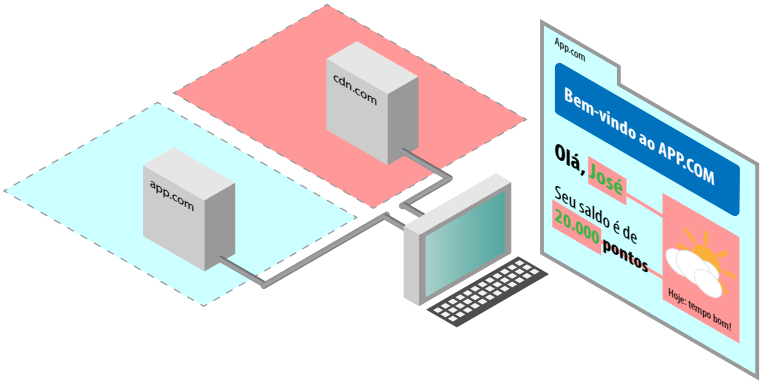

Figura 4 – Domínio de CDN comprometido, capturando informações do usuário.

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).

2.1.3.3 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) (QUAIS?) 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.

2.1.3.4 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 ter-

ceiros 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.

2.1.4 Aparato de segurança da informação implementado pelos navegadores

2.1.5 Requisitos para avaliação da segurança da informação no navegador

2.1.6 Abordagens para mitigação de riscos

2.1.6.1 Políticas padronizadas

2.1.6.1.1 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, 2017b).

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.

2.1.6.1.2 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.

2.1.6.1.3 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).

2.1.6.1.4 SRI – Subresource Integrity

2.1.6.2 Abordagens proprietárias

2.1.6.2.1 Abordagens experimentais

2.1.6.2.2 JSFLow

2.1.6.2.3 SessionGuard

2.1.6.2.4 Abordagens em processo de padronização

2.1.6.2.5 COWL

2.1.6.2.6 CSP Nível 3

2.1.7 Processo de renderização do documento web

Para que seja visualizada no navegador, o conteúdo da página web passa pelo processo de renderização implementado pelo mecanismo de visualização. Nesse processo, o navegador leva em consideração características ambientais como a resolução do dispositivo gráfico, o tamanho do *viewport* (janela do navegador), o tipo de dispositivo de saída (tela, impressora) e os recursos de sistema disponíveis (taxa de utilização da CPU, do processador gráfico e da memória RAM), e também definições da página web como o conjunto de folhas de estilo, recursos audiovisuais incorporados (imagens, vídeos) e o próprio conteúdo textual codificado em marcação HTML. Dado que quaisquer dessas características pode variar durante o ciclo de vida de uma página web, o navegador pode repetir o processo de renderização diversas vezes em resposta a essas variações.

GRIGORIK (2018), ao descrever o processo de renderização implementado pelo navegador Chrome, menciona duas etapas iniciais: a construção da árvore de objetos do HTML, resultando na estrutura DOM, e a construção da árvore de definições de CSS, resultando na estrutura CSSOM. Ambas as estruturas são independentes entre si, mas sua combinação tem efeito sobre a terceira etapa da renderização, responsável por construir a árvore de visualização (*render tree*) da página web.

A árvore de visualização fornece ao navegador a informação necessária para a definição do *layout* de cada elemento HTML em função da visibilidade, das dimensões e do estilo de cada elemento. O *layout* é um dado de entrada para o processo de desenho que gera *pixels* visíveis no dispositivo de saída. Uma síntese desse processo é enumerada por GRIGORIK (2018) nos seguintes passos, ilustrados pelo diagrama 2.1.7:

a) O DOM e o CSSOM são combinados na árvore de visualização;

b) É calculado o *layout* – tamanho, posição e estilo – de cada elemento;

c) Elementos sem *layout* são descartados da árvore;

d) Os nós da árvore de visualização são desenhados no dispositivo de saída.

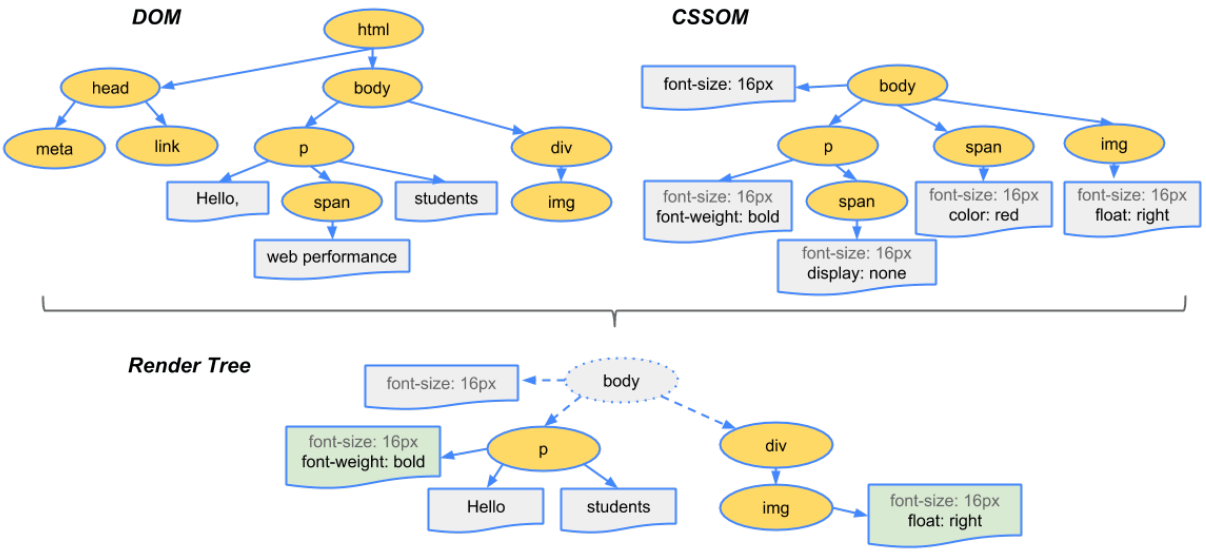

Figura 5 – Construção da árvore de visualização do navegador (GRIGORIK, 2018)

2.1.7.1 Conflitos de regras em CSS

A estrutura CSSOM deriva da combinação das regras (*style rules*) de todas as folhas de estilo incorporadas pela página. A existência de um CSSOM único pode desencadear efeitos colaterais como a colisão de nomes de regras (simultaneamente definidas em folhas de estilo concorrentes) e a herança inesperada de atributos de estilo, ambos levando a *layouts* diferentes daquele esperado pelo desenvolvedor de uma página web. O diagrama 2.1.7.1 exemplifica esses efeitos.

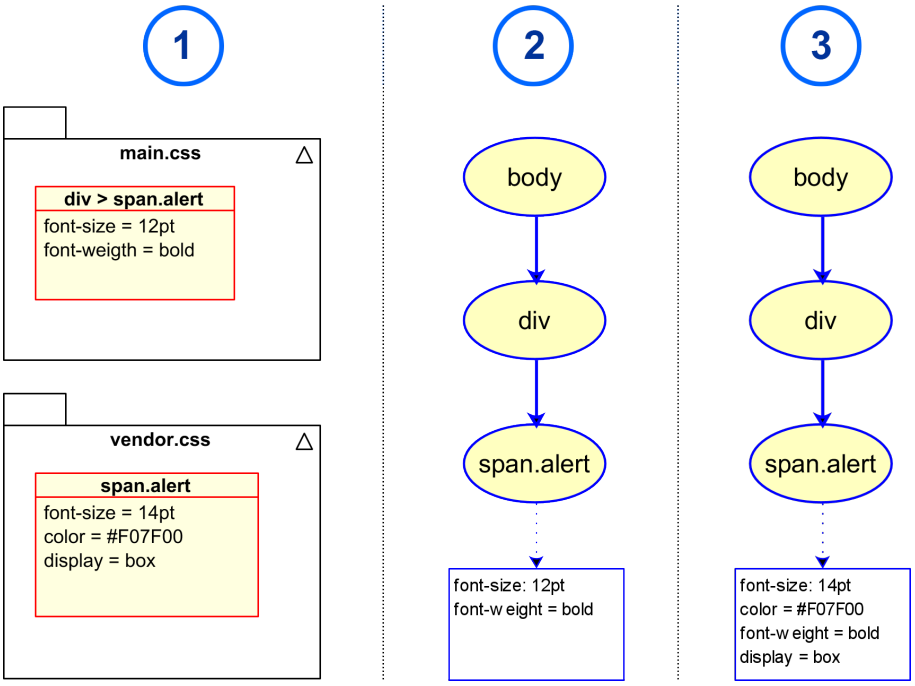

Figura 6 – Conflito de regras em folhas de estilo. Sob (1), duas folhas de estilo independentes, **main.css** e **vendor.css**, são incorporadas a uma mesma página. O desenvolvedor da página não tem controle sobre **vendor.css**, que inadvertidamente redefine a regra de estilos aplicável aos elementos **span** decorados com a classe **alert**. Com isso, a estrutura CSSOM pretendida pelo desenvolvedor da página, sob (2), é calculada de modo a produzir um layout divergente, ilustrado pela estrutura CSSOM sob (3).

Como o meacnismo de visualização é tolerante aos conflitos de especificação de regras de estilo, as soluções para os efeitos colaterais decorrentes é paliativa. Os desenvolvedores são encorajados a estabelecer “espaços de nome” que delimitem o escopo de aplicação das regras de estilo[2], diminuindo a chance de que as árvores CSSOM resultantes contenham nós conflitantes. No entanto, uma proposta mais robusta, inaugurada pela padronização de *shadow DOMs*, pode ser empregada com o propósito de isolar árvores de CSSOM e, consequentemente, produzir árvores de visualização independentes.

2.1.7.2 Isolamento de árvores de visualização com *Shadow DOM*

CONTINUAR DEPOIS. Falar sobre os conceitos: shadow host, shadow tree. Não falar sobre: modos open e closed.

[2] “Defensive HTML and CSS” – <https://www.webdesignerdepot.com/2013/04/defensive-html-and-css/>

A proposta do mecanismo de Shadow DOM (W3C, 2017b) é definir uma forma padronizada para que o próprio navegador minimize as interferências entre componentes de página que, de outra forma, poderiam causar as já mencionadas colisões de regras de 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.

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.

2.2 Estado da arte

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

2.2.0.1 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.

2.2.0.2 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.

2.2.0.3 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.

2.2.0.4 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.

2.2.0.5 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.

2.2.0.6 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.

2.3 Avaliação das abordagens para segurança da informação no navegador

2.3.1 Cenários de risco

A revisão bibliográfica evidencia algumas das abordagens contra o vazamento de informação provocado por *scripts* atuando de forma anômala. Justifica-se, contudo, o questionamento de quão disseminados estão os riscos que essas abordagens visam mitigar. Uma forma de responder a esse questionamento é a avaliação empírica de situações, ou cenários, onde o usuário possa se encontrar em risco de ter informação confidencial submetida a vazamento.

3 PROPOSTA

Esta dissertação propõe um mecanismo discricionário contra vazamento de informação contida no DOM baseado em duas capacidades: (1) impedir que *scripts* encontrem nós específicos do DOM via APIs de manipulação de árvore[1] e (2) impedir que *scripts* sejam notificados da ocorrência de eventos do DOM originários de nós específicos. Essas capacidades efetivamente blindam a informação contida em nós do DOM selecionados.

As próximas seções encarregam-se de apresentar o mecanismo em termos de sua especificação e de seu funcionamento, descrevendo as tecnologias em que ele é fundamentado e sua aptidão frente aos casos de uso levantados na seção X.

3.1 API de ocultação do DOM fundamentada em Shadow DOM

Com base no isolamento de árvores de DOM proporcionado pelo recurso de *shadow DOM*, descrito preliminarmente em 2.1.7.2, é possível explorar o comportamento derivado de uma propriedade das APIs que estabelece, em sua especificação, que para todo *shadow host* será determinada a visibilidade de sua *shadow tree*. O conceito de *visibilidade*, neste contexto, corresponde à possibilidade de que *scripts* consigam alcançar nós do DOM contidos na *shadow tree*, tanto indiretamente, enumerando os nós-filhos da árvore por meio da propriedade `ShadowRoot.childNodes`, quanto diretamente, por meio de métodos como `getElementById()` ou `querySelector()`, implementados por objetos como `Element`, `HTMLDocument` e `DocumentFragment`.

O atributo de visibilidade é determinado no momento da criação do *shadow host*. A listagem de código 3.1, entre as linhas 2 e 6, exemplifica os efeitos mais evidentes desse atributo. Essencialmente, os *scripts* ganham ou perdem acesso à propriedade `Element.shadowRoot`, na medida em que o parâmetro `mode` é determinado como `open` ou `closed`.

```
1  var openShadowHost = document.createElement('div');
```

[1] Essas APIs incluem métodos e propriedades comuns do "DOM Core", como `document.getElementById()`, `ParentNode.querySelector()` e `Element.childNodes`.

```
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

Tornando inacessível a propriedade `Element.shadowRoot`, o método `Element.attachShadow({mode:'closed'})` provê um meio programático para a definição de regiões protegidas na estrutura DOM da página web. No entanto, a inacessibilidade do conteúdo da *shadow root* depende de medidas que vão além das garantias previstas pela especificação da API de Shadow DOM. Isso se deve à extensibilidade da linguagem Javascript, que permite que *scripts* interceptem quaisquer métodos de qualquer API. Desse modo, um *script* cuidadosamente escrito poderia ganhar acesso a todas as *shadow roots* criadas em uma página web, independentemente da opção `open` ou `closed` atribuída ao parâmetro `mode` em `attachShadow()`. Essa estratégia é detalhada na seção seguinte.

3.1.1 Interceptação de métodos nativos do ambiente de execução

A linguagem Javascript apresenta *herança prototípica* como mecanismo para compartilhar estrutura, comportamento e estado entre os objetos que participam de um mesmo relacionamento denominado de *cadeia prototípica*. Por esse mecanismo, a linguagem Javascript admite o uso de abstrações de classes, a exemplo de linguagens estáticas como Java e C++ (ECMA International, 2017, seção 4.2.1). Porém, ao contrário dessas linguagens, Javascript possibilita a redefinição dinâmica dos membros dos

protótipos/classes. A listagem 3.2 exemplifica esse mecanismo.

```
var myClass = {
  nomeAtribuido: '',
  idade: 0,
  exibirNome: function() {
    console.log('Meu nome é ' + this.nomeAtribuido);
  }
};

// A função Object.create() cria um novo objeto
// baseado em um protótipo, como na linha a seguir:
var objetoHerdeiro = Object.create(myClass);
objetoHerdeiro.nomeAtribuido = 'Teste';

// objetoHerdeiro herda as funções definidas em myClass e redefine o valor
// do membro nomeAtribuido). A função (*exibirNome, por exemplo, faz referência
// a esse membro, inicializado como string vazia em myClass, mas retornado
// como 'Teste' na chamada de método a seguir:
objetoHerdeiro.exibirNome();

// O objeto myClass ganha um novo membro, a função exibirIdade:
myClass.exibirIdade = function() {
  console.log('Minha idade é ' + this.idade);
};

// A função recém-definida no protótipo é imediatamente disponível
// como membro herdado:
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 substituído ou encapsulado por código arbitrário. Essa capacidade de interceptação permite que métodos sejam instrumentados, monitorados ou mesmo corrompidos sem que o *runtime* da linguagem levante impedimentos. 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".

```
// A variável number contém uma constante de valor numérico,
// instância derivada do prototype Number:
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 redefinição do método toString() do objeto Number

Virtualmente qualquer dos membros de objetos em Javascript, incluindo propriedades e métodos dos objetos primitivos `Boolean`, `Number`, `String` e `Object`, podem ser redefinidos em suas cadeias de herança. MAGAZINIUS et al. (2012), em seu estudo sobre a efetividade das estratégias empregadas por *sandboxes* como Google Caja e Facebook Javascript, denomina a redefinição mal-intencionada da cadeia de herança como "envenenamento do protótipo" (*prototype poisoning*, no artigo original). É por esse motivo que a mera utilização de `{mode: 'closed'}` como parâmetro do método `Element.attachShadow()` não é suficiente para tornar invioláveis as *shadow roots* que forem criadas com a intenção de serem "ocultas", pois a redefinição desse método, exemplificada pela listagem 3.4, tem acesso todas as referências às *shadow roots* criadas durante o ciclo de vida de uma página, podendo ler e modificar o conteúdo tido como protegido pelo desenvolvedor.

```
1  // Uma referência ao método attachShadow, do prototype Element,
2  // é armazenada na variável metodoInterceptado:
3  var metodoInterceptado = Element.prototype.attachShadow;
4
5  // É definido um array para posterior armazenamento das shadow roots
6  // que forem criadas, independentemente da opção de visibilidade atribuída pelos desenvolvedores
       .
7  var everyShadowRoot = [];
8
9  // O método attachShadow é redefinido no prototype Element.
10 // Deste ponto em diante, todos os elementos HTML referenciados no DOM passam a
11 // refletir o método redefinido.
```

```
12 Element.prototype.attachShadow = function sniffer() {
13   // O método original, nativo do navegador, é aplicado para que a shadow root
14   // seja efetivamente criada; a coleção arguments contém o parâmetro
15   // mode de valor 'open' ou 'closed'.
16     var newShadowRoot = metodoInterceptado.apply(this, arguments);
17
18     // A referência à nova shadow root é armazenada para uso (e abuso) posterior.
19     everyShadowRoot.push(newShadowRoot);
20
21     return newShadowRoot;
22 };
```

Código 3.4 – Exemplo de interceptação do método Element.attachShadow()

3.1.1.1 Uma estratégia para neutralizar a redefinição de APIs

No *runtime* de Javascript, a ordem de declaração dos *scripts* determina sua sequência de execução, a qual ocorre em série – é garantido que apenas um *script* seja executado em um dado momento (ECMA International, 2017, seção 8.3). Esse comportamento sustenta a elaboração de uma estratégia para mitigação dos riscos inerentes à interceptação dinâmica dos métodos em Javascript: o primeiro *script* a ser executado poderia armazenar referências aos métodos que lhe forem importantes, em uma rotina de *setup*. Desta forma, mesmo se os *scripts* carregados subsequentemente efetuarem sobrescrita de protótipos, as implementações originais dos métodos requeridos estarão a salvo da manipulação. Para tanto, as seguintes garantias precisam ser obedecidas:

a) Garantia de que um *script* confiável de *setup* seja o primeiro bloco de código Javascript carregado pelo navegador na página web.

b) Garantia de que os métodos de que o desenvolvedor necessite para interação com informações sensíveis sejam referenciados por variáveis locais do *script* de *setup*.

c) Garantia de que as chamadas aos métodos referenciados ocorram por via indireta, ignorando os `prototypes` de que fizerem parte.

A terceira garantia depende da invocação de funções por meio do método `Function.apply(thisObj, methodFn)` em detrimento de chamadas diretas como `thisObj.methodFn()`. A listagem 3.5 fornece um exemplo de implementação dessa estratégia.

```
var number = 41,
  nativeToString = Number.prototype.toString;

console.log('Antes da manipulação de Number.toString() - ' + number.toString());

// O método toString() será adulterado para que sempre retorne '42'
// para qualquer instância de Number:
Number.prototype.toString = function () { return '42'; };

// A chamada a seguir produzirá o resultado incorreto '42':
console.log('Depois da manipulação de Number.toString() - ' + number.toString());

// Porém, a chamada a seguir produzirá '41', refletindo o valor subjacente correto:
console.log('Chamada imune à manipulação ocorrida - ' + nativeToString.apply(number, []));
```

Código 3.5 – Implementação da estratégia de chamada indireta de métodos

3.1.1.2 Falhas na estratégia de neutralização

Ainda que o algoritmo de neutralização apresente certa capacidade de neutralizar os riscos de envenenamento do protótipo de objetos, sua implementação está sujeita a outros riscos. Em primeiro lugar, a garantia de que determinado *script* seja carregado antes dos demais é, na prática, frágil. Os *scripts* interessados nessa proteção podem não estar sob controle do autor de uma aplicação web, especialmente se fizerem parte de soluções de terceiros cuja ordem de carregamento e execução seja imprevisível *a priori*. Ademais, em segundo lugar, é plenamente possível que um *script* mal-intencionado tome controle exatamente do método `Function.apply()`, o que tornaria vulneráveis todas as chamadas de função dependentes desse método. A listagem 3.6 evidencia essa modalidade de interceptação, demonstrando a relativa simplicidade de se contornar a estratégia proposta.

```
// Início dos blocos de script na página web.
// ------ <script src="safe-attachShadow.js">
// Uma referência ao método attachShadow (nativo e confiável) é armazenada
// na variável safeMethodRef. O método referenciado será invocado pela
// função createClosedShadowRoot(), exportada por este script.
var safeMethodRef = Element.prototype.attachShadow;
var createClosedShadowRoot = function(host) {
  return safeMethodRef.apply(host, [{mode: 'closed'}]);
};
// </script>
```

```
11
12 // ------ <script src="stealthy-apply.js">
13 // Este script, oriundo de um componente de terceiros incorporado à página,
14 // vai interceptar todas as chamadas ao método Function.apply.
15 var nativeApply = Function.prototype.apply;
16 nativeApply.apply = nativeApply; // Esta atribuição impede a recursividade infinita
17                                  // em chamadas a Function.apply()
18
19 Function.prototype.apply = function() {
20   var res = nativeApply.apply(this, arguments);
21   if (res instanceof ShadowRoot) {
22     res.innerHTML = 'Shadow root interceptada';
23   }
24
25   return res;
26 }
27 // </script>
28
29 // ------ <script src="app.js">
30 // Este script fará uso da função createClosedShadowRoot(), sem que o desenvolvedor tenha
31 // conhecimento de que a shadow root retornada será visível a outros scripts.
32 var host = document.createElement('div'),
33     sr = createClosedShadowRoot(host);
34
35 document.body.appendChild(host);
36 // </script>
37
38 // Neste momento, o elemento host e sua shadow root estão incorporados ao
39 // DOM. E, por causa do comportamento do script stealthy-apply.js,
40 // a mensagem "Shadow root interceptada" será visível na tela.
```

Código 3.6 – Subversão do método Function.apply() como meio de ganhar acesso às *shadow roots* fechadas

Pelas fragilidades apontadas, uma outra estratégia deve ser proposta – uma que seja mais tolerante quanto às garantias necessárias, porém mais robusta em ambientes de execução hostis.

3.1.2 Detectando a interceptação de métodos em Javascript

Em vista das características dinâmicas da linguagem Javascript, e em consequência à tolerância que o navegador demonstra para com todos os *scripts* incorporados em uma página web, este trabalho toma como premissa que não é factível estabelecer

um ambiente de execução imune às diversas maneiras nas quais as APIs de Javascript podem ser exploitadas, especialmente quando o objetivo do agente mal-intencionado é facilitar o vazamento de informação. Ao contrário, este trabalho propõe uma abordagem que pode contribuir para que as aplicação web não sejam expostas aos riscos derivados de *scripts* que interfiram na cadeia de herança prototípica. Por meio dessa abordagem, a aplicação determina, tão cedo quanto possível no ciclo de vida da página, se as APIs de que depende foram de algum modo comprometidas e, em caso positivo, que nenhuma informação confidencial seja exposta. Esta é uma estratégia de *early fail*: se determinadas condições não forem atendidas na inicialização da página, que nenhuma operação posterior tenha a oportunidade de ser executada.

Para tanto, é empregado um dispositivo de software que detecta a ocorrência de interceptação de métodos em Javascript. Esse dispositivo corresponde a um processo capaz de avaliar, em tempo de inicialização, se determinados métodos de API tiveram sido substituídos por interceptadores, o que para os propósitos do dispositivo os caracteriza como não-confiáveis. A existência de ao menos um método não-confiável é suficiente para que o dispositivo acuse uma falha de inicialização, a partir do que a aplicação web poderá desviar o fluxo para uma condição de erro. A não-ocorrência de falha, por sua vez, significa que os métodos avaliados não foram interceptados, sendo considerados como confiáveis. O diagrama 3.1.2 ilustra a sequência de atividades desempenhadas durante o processo de detecção de métodos não-confiáveis.

[DIAGRAMA 3.1.2]

A implementação do dispositivo de detecção se baseia na conversão implícita de valores passados como argumento para os métodos avaliados. O dispositivo é capaz de detectar interceptação em métodos que esperam ao menos um parâmetro do tipo `String` e que, para tais parâmetros que receberem como argumentos valores de outros tipos, o método tente convertê-los em *strings*. Em Javascript, todo objeto herda do protótipo `Object` o método `toString()` (ECMA International, 2017, seção 19.1), cujo propósito é criar uma representação textual do valor intrínseco de um objeto. Embora esse propósito seja distinto de uma “conversão de tipo”, este é o efeito observado em situações como a do seguinte trecho de código:

```
var o = { toString: function () { return 'Hello, world'; } };
document.write(o);
// A expressão "Hello, world" terá sito escrita na página web
```

Implicitamente o método `document.write()` "converte" o argumento `o` em um valor `String` por meio da chamada de seu método `toString()` (o qual, de fato, sobrescreve o método `toSring()` herdado de `Object` através da cadeia prototípica). Esse padrão de conversão é empregado em outras APIs de Javascript e também em operações como a de concatenação de *strings*:

```
var s = o + ' - have a nice day!';
// O conteúdo da variável s é "Hello, world - have a nice day"
```

Para fins do dispositivo de detecção de interceptação, um objeto não-textual (denominado *vetor*) é passado como argumento para os métodos sob avaliação (denominados *suspeitos*) e estes, ao realizarem a conversão do vetor em `String`, irão implicitamente acionar o método `vetor.toString()`. Mais do que devolver uma representação textual do vetor, a implementação de `toString()` tem a responsabilidade de obter informações sobre o contexto de execução do método suspeito e, a partir disso, inferir se esse contexto pertence a uma interceptação de API. Essa inferência se baseia na profundidade da pilha de chamadas (*call stack*) em termos do número de chamadas aninhadas (*stack frames*). O dispositivo emprega uma heurística cuja premissa é de que deve existir uma número predeterminado de *stack frames* entre a chamada de `vetor.toString()` e a chamada do próprio método suspeito. Os experimentos demonstram que a profundidade de pilha esperada para todos os métodos avaliados é 1: um número maior de *frames* significa que há mais chamadas em curso, caracterizando a interceptação do método suspeito. A listagem 3.7 demonstra a implementação de vetor utilizada nos experimentos deste trabalho.

```
function countFrames(stackTrace) {
    return !stackTrace && 0 || stackTrace.match(/(\n)/mg).length;
}

function buildVector() {
    let stackTrace = new Error().stack,
        outerFrameCount = countFrames(stackTrace),
        innerFrameCount,
```

```
9           flagged = false;
10
11      return {
12          toString: function () {  // => Definição do método vetor.toString(), que implementa
13                                   //    a avaliação da profundidade da pilha de chamadas
14                                   //    para determinar se um método de API foi interceptado
15              innerFrameCount = countFrames(new Error().stack);
16              flagged = (innerFrameCount - outerFrameCount) > 0;
17              return '';
18          },
19          isFlagged: function () { // => Definição do método vetor.isFlagged(), que retorna
20                                   //    o resultado da avaliação de interceptação
21              return flagged;
22          }
23      };
24  }
```

Código 3.7 – Implementação do vetor de detecção de métodos interceptados

O método `vetor.toString()`, núcleo do dispositivo de detecção, avalia a profundidade da pilha de chamadas pela inspeção da propriedade `stack` de um objeto `Error`. Embora não seja padronizada, a propriedade `Error.stack` apresenta comportamento consistente entre os mecanismos de execução de Javascript, retornando uma sequência de linhas de texto correspondentes às *stack frames* – um efeito demonstrado na listagem 3.8. Assim, a contagem de linhas de texto contidas no *stack trace* é o resultado tomado como profundidade da pilha.

Experimentos evidenciam que, em diferentes plataformas[2], não é possível substituir a implementação nativa da propriedade `Error.stack`. Neste trabalho, foram desenvolvidos *scripts* que tentaram, sem sucesso, interceptar e modificar o conteúdo de `Error.prototype.stack`, além da substituição do próprio construtor e protótipo do objeto `Error`. Independentemente dessas tentativas, os *runtimes* de Javascript reforçam uma característica de previsibilidade no comportamento do objeto `Error`. Essa constatação dá segurança para que o vetor utilizado pelo dispositivo seja considerado confiável em sua capacidade de detectar variações na pilha de chamadas.

```
1  var a = function () { console.log(new Error().stack); };
2  var b = function() { a(); };
```

[2] NodeJS v8.9.4; Chrome 66; Microsoft Edge estão em conformidade entre si; notável exceção do Firefox Quantum 59.

```
3  var c = function() { b(); };
4
5  // Resultados de chamadas no console do navegador Chrome.
6  // A chamada à função a() resultará em um ”stack trace” de 3 linhas:
7  a();  // => VM66:1 Error
8        //        at a (<anonymous>:1:31)
9        //        at <anonymous>:1:1
10
11 // A chamada à função b() resultará em um ”stack trace” de 4 linhas:
12 b();  // => VM66:1 Error
13       //        at a (<anonymous>:1:31)
14       //        at b (<anonymous>:1:18)
15       //        at <anonymous>:1:1
16
17 // A chamada à função c() resultará em um ”stack trace” de 5 linhas:
18 c();  // => VM66:1 Error
19       //        at a (<anonymous>:1:31)
20       //        at b (<anonymous>:1:18)
21       //        at c (<anonymous>:1:18)
22       //        at <anonymous>:1:1
```

Código 3.8 – Diferentes conteúdos de Error.stack quando se consideram aninhamentos mais profundos de chamadas de função

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

3.3 Apresentação do site acadêmico “js-infosec.info”

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.

BIELOVA, N. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. **Journal of Logic and Algebraic Programming**, v. 82, n. 8, p. 243–262, 2013. ISSN 15678326.

BRIGHT, P. **Firefox takes the next step toward rolling out multi-process to everyone**. 2016. Disponível em: <https://arstechnica.com/information-technology/2016/12/firefox-takes-the-next-step-towards-rolling-out-multi-process-to-everyone/>.

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>.

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/>.

ECMA International. Standard Ecma-262. p. 885, 2017. Disponível em: <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.

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>.

GRIGORIK, I. **Constructing the Object Model**. 2018. Disponível em: <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model>.

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/c073906_ISO_IEC_27000_2018_E.zip>.

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.

MAGAZINIUS, J. et al. Safe wrappers and sane policies for self protecting JavaScript. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 7127 LNCS, p. 239–255, 2012. ISSN 03029743.

NGUYEN, N. **The Best Firefox Ever**. 2017. Disponível em: <https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/>.

NIKIFORAKIS, N. et al. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. **Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12**, p. 736, 2012. ISSN 15437221. Disponível em: <http://dl.acm.org/citation.cfm?doid=2382196.2382274>.

OWASP. **DOM Based XSS**. 2015. Disponível em: <https://www.owasp.org/index.php/DOM_Based_XSS>.

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

____. **Cross Site History Manipulation (XSHM)**. OWASP, 2017. Disponível em: <https://www.owasp.org/index.php/Cross_Site_History_Manipulation_(XSHM)>.

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

____. **DOM Based XSS**. 2017. Disponível em: <https://www.owasp.org/index.php/Top_10-2017_Top_10>.

PATIL, K. Isolating malicious content scripts of browser extensions. **International Journal of Information Privacy, Security and Integrity**, v. 3, n. 1, p. 18, 2017. ISSN 1741-8496. Disponível em: <http://www.inderscience.com/link.php?id=10007834>.

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>.