TaintJSec: Um Método de Análise Estática de Marcação em Código JavaScript para Detecção de Vazamento de Dados Sensíveis

Alexandre Damasceno[1], Thiago Rocha[2], Eduardo Souto[2]

[1]Samsung Instituto de Desenvolvimento para a Informática da Amazônia – Manaus, AM – Brasil

[2]Instituto de Computação
Universidade Federal do Amazonas (UFAM) – Manaus, AM – Brasil

alexandre.d@samsung.com, {thiago.rocha,esouto}@icomp.ufam.edu.br

Abstract. *JavaScript is one of the most widely used programming languages in the world because it is highly dynamic. However, the same feature that makes JavaScript a successful programming language makes it difficult to perform static code analysis to identify the presence of malicious code. This article introduces TaintJSec, an approach that uses static JavaScript markup parsing to detect leakage of sensitive information. The TaintJSec analyze the flow of implicit codes, performs the propagation of the taint tag in the eval function and identifies the information leakage in obfuscated codes. The tests realized demonstrated that the approach is effective in detecting information leakage and more efficient than other methods of the state of the art.*

Resumo. *JavaScript é uma das linguagens de programação mais utilizadas no mundo por ser altamente dinâmica. Porém, essa mesma característica que a torna uma linguagem de sucesso torna difícil a realização de análise estática de código para identificar a presença de códigos maliciosos. Este artigo apresenta o TaintJSec, uma abordagem que usa análise estática de marcação de código JavaScript para detectar o vazamento de informação sensível. O TaintJSec analisa o fluxo de códigos implícitos, propaga a taint tag na função eval e identifica o vazamento de informação em códigos ofuscados. Os testes realizados demonstraram que a abordagem é eficaz na detecção do vazamento de informação e mais eficiente que outros métodos do estado da arte.*

1. Introdução

A linguagem *JavaScript* é uma das linguagens de programação mais utilizadas no mundo para a criação de aplicações *web* (Fard et al. 2017). Esse sucesso é decorrente da sua facilidade de manipulação, pois a linguagem permite que um mesmo trecho de código possa ser escrito de diferentes formas e da alta dinamicidade da linguagem que possibilita a inclusão de trechos de código em tempo de execução (Ocariza et al. 2013), permitindo que aplicações desenvolvidas em *JavaScript* executem rotinas de código e percorram fluxos de execução não explicitamente conhecidos.

Essa versatilidade, embora seja útil para o desenvolvimento de aplicações, acaba abrindo um leque de oportunidades para a execução de códigos maliciosos, que podem

vir a comprometer os pilares da segurança da informação como a confidencialidade e a integridade dos dados.

Como a linguagem *JavaScript* vem evoluindo continuamente e ganhando novos campos de atuação como o desenvolvimento de extensões para navegadores (Google 2018; Mozilla 2018), aplicativos *desktop* (WinJS 2018; Electron 2018), atuação em servidores (Node.js 2018) e em dispositivos móveis (Tizen 2018), o interesse pela criação de soluções que pudessem resolver os problemas de privacidade decorrentes do vazamento de informação vem ganhando força.

O vazamento de informação é a distribuição acidental ou não intencional de dados sigilosos para um destino não autorizado (Peneti et al. 2016). Esse problema ocorre com maior incidência no universo dos dispositivos móveis (Gibler et al. 2012), onde é explorado por aplicações maliciosas que, uma vez instaladas nos dispositivos, capturam dados sigilosos para repassá-los a terceiros.

Na literatura, diversos trabalhos tentam coibir o problema do vazamento de informação sensível usando diferentes abordagens. Alguns trabalhos utilizam análise do fluxo de dados e verificam, por exemplo, o conteúdo de pacotes HTTP para checar a existência de informação sigilosa (Pinto et al. 2016) e (Kuzuno et al. 2013). Outros trabalhos utilizam listas para permitir ou bloquear o acesso às fontes de informação sensível (Wang et al. 2014) e (Hsiao et al. 2014). Entretanto, a maioria dos trabalhos realiza análise estática ou dinâmica do código para observar o fluxo de funcionamento de uma aplicação e descobrir se uma informação sensível está sendo vazada (Theriault 2013; Patnaik et al. 2015; Monteiro et al. 2013).

Independentemente da abordagem utilizada, existem muitos desafios que precisam ser superados. Os trabalhos que procuram detectar o vazamento mediante análise de pacotes HTTP não conseguem identificar o vazamento se os dados estiverem ofuscados. O uso de listas para permitir ou bloquear o acesso às fontes de informação sensível exigem que o usuário tenha conhecimento técnico o suficiente para decidir o que deve ser permitido e o que deve ser bloqueado. Enquanto os trabalhos que utilizam análise de código esbarram na complexidade computacional do código *JavaScript* devido a sua versatilidade. A dificuldade para analisar a execução da função *eval,* por exemplo*,* é o fator determinante para que muitos trabalhos de análise estática de código *JavaScript* não deem suporte à análise de criações dinâmicas de código.

Para superar estes problemas, este artigo apresenta o *TaintJSec,* uma abordagem que utiliza a análise estática para identificar e prevenir o vazamento de informação sensível em aplicações *web*. O *TaintJSec* é capaz de identificar o vazamento de informação sensível em códigos implícitos, como também em códigos ofuscados. Além disso, este trabalho propõe um novo método de marcação de informações, denominado *taint position*, que permite aumentar o controle da propagação da *taint tag* na análise da função *eval*. Os resultados obtidos demonstraram que a abordagem é eficaz na detecção do vazamento de informação e mais eficiente que outros métodos do estado da arte.

O restante desse trabalho está organizado da seguinte forma: A Seção 2 descreve as principais abordagens para resolver o problema do vazamento de dados sensíveis. Seção 3 fornece uma visão geral do *TaintJSec*. A Seção 4 apresenta os experimentos realizados e resultados obtidos. Por fim, a Seção 5 conclui o artigo e expõe alguns trabalhos futuros.

2. Trabalhos Relacionados

Esta seção discute as principais abordagens para resolver o problema do vazamento de informação sensível.

Alguns trabalhos empregam técnicas de inspeção de pacotes para detectar vazamento de informações (Kuzuno et al. 2013), (Pinto et al. 2016). Por exemplo, Kuzuno et al. propõem uma abordagem baseada na análise de pacotes que utiliza técnicas de clusterização para detectar o vazamento de dados. O agrupamento de dados do tráfico de rede é utilizado para criar assinaturas. Os autores assumem como premissa que os clusters formados por pacotes contendo dados sensíveis são diferentes daqueles que não possuem informação sensível.

Outros trabalhos detectam vazamento de dados com base na utilização de listas. Hsiao et al. (2014) e Wang et al. (2014) utilizam listas para controlar quais aplicações podem ter acesso a dados sensíveis. Wang et al. propõem uma abordagem que é configurável e permite que o usuário possa informar quais aplicações podem acessar dados sensíveis. Todas as aplicações cadastradas na lista passam a fornecer valores nulos para informações cadastradas como sensíveis como o número do telefone, ID do dispositivo e IMEI, entre outras.

Na maioria dos trabalhos da literatura, a detecção de vazamento de dados é baseada no emprego de técnicas de análise de código (estática ou dinâmica) da aplicação. Kashyap et al. (2014) propõem um analisador de código *JavaScript* que utiliza uma árvore sintática abstrata (AST) para percorrer o código e realizar a análise de fluxo. Os autores validam a AST por meio de uma comparação com a ferramenta comercial SpiderMonkey. A avaliação dos resultados, usando 243 casos de teste, mostra que analisador proposto é capaz de identificar a maioria dos casos avaliados. Entretanto, a abordagem não é capaz de analisar a execução da função *eval* ou de qualquer outra inserção de código dinâmico como, por exemplo, a criação de funções utilizando o objeto *new Function*.

Fard et al. (2013) propõem o JSNose, um analisador híbrido para realizar a inspeção do código e identificar trechos de código *JavaScript* em um programa. O JSNose procura blocos de código como instrução de *switch* vazias, *catch* sem argumentos e outras situações que servem de apoio ao desenvolvedor. A análise estática é realizada utilizando Rhino, que também realiza a análise sintática e fornece a AST. Semelhante ao trabalho de Kashyap et al., o JSNose não é projetado para analisar a execução da função *eval* e outras criações dinâmicas de código.

Patnaik et al. (2015) desenvolveram uma ferramenta de análise estática de código denominada *JSPrime*, que verifica o fluxo de execução para encontrar vazamento de informação. Assim como o *TaintJSec*, o JSPrime utiliza um código intermediário para realizar a verificação do fluxo de código. Porém, seu analisador não verifica códigos gerados a partir da função *eval*. Em vez disso, o JSPrime trata a função *eval* como um *taint sink*.

Theriault (2013) propôs a ferramenta *ScanJS* para realizar análise estática de código baseada em padrões de assinatura geradas a partir da análise de uma representação intermediária do código em formato de AST. A AST é percorrida em busca dos padrões de assinatura previamente definidos pela ferramenta e o resultado da análise é apresentado ao usuário.

Monteiro et al. (2013) propõem a ferramenta *JSpwn* para realizar análise estática no código *JavaScript*. *JSPwn* é uma versão modificada do *JSPrime* que une recursos do *ScanJS* para detectar vulnerabilidades. Com o mecanismo do *ScanJS* para detectar vulnerabilidades e o recurso de análise de fluxo de código do *JSprime*, a ferramenta tem a capacidade de detectar pontos de vulnerabilidades ao percorrer o código.

Diferentemente dos trabalhos mencionados, este trabalho propõe uma solução para lidar com os problemas de verificação de códigos dinâmicos, como trechos de código construídos usando a função *eval*. Além disso, o método proposto é capaz de detectar vazamento a partir de códigos ofuscados.

3. TaintJSec

Esta Seção apresenta o *TaintJSec*, uma abordagem de análise estática de marcação de código JavaScript para identificar e prevenir o vazamento de informação. A Figura 1 mostra uma visão geral do *TaintJSec* que consiste das etapas de: 1) identificação e extração do código *JavaScript* que estão declarados explicitamente na aplicação; 2) representação do código-fonte extraído em uma Árvore de Sintática Abstrata (AST – *Abstract Sintax Tree*); 3) um analisador do fluxo de informação que visita todas as entidades, objetos, propriedades e funções, representadas na AST, para analisar suas estruturas e relações com o objetivo de identificar a ocorrência de um vazamento de dados.

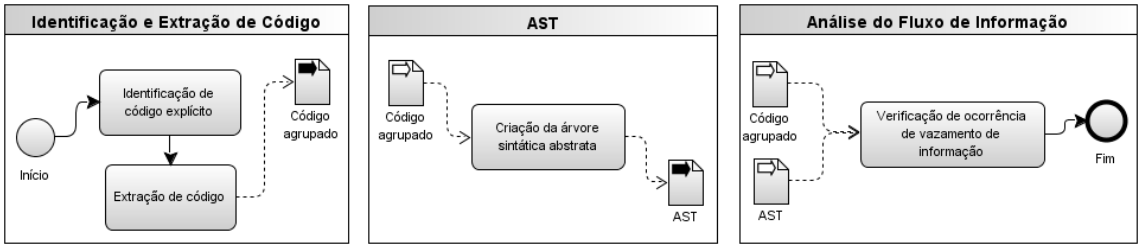

Figura 1. Visão geral das etapas que compõem o analisador TaintJSec

3.1. Identificação e Extração de Código

A primeira fase do *TaintJSec* tem como objetivo identificar e extrair todos os códigos *JavaScript* que estão declarados explicitamente na aplicação, separando-os do restante do código HTML e os organizando de acordo com a ordem em que aparecem, sem alterar o fluxo de execução das instruções. Dessa forma, o fluxo da informação presente no código agrupado é semelhante ao fluxo original presente na aplicação.

Quando um código válido é encontrado, todas as suas particularidades são preservadas. Nenhum comentário é removido, assim como nenhuma tentativa de minificar o código é executada. Isso é importante para que o analisador utilize um código *JavaScript* idêntico ao presente na aplicação. A Figura 2 apresenta uma representação do processo de extração do *TaintJSec*.

A primeira fase do *TaintJSec* termina quando todos os códigos *JavaScript* existentes na página HTML tiverem sido extraídos e devidamente organizados formando um só bloco de código.

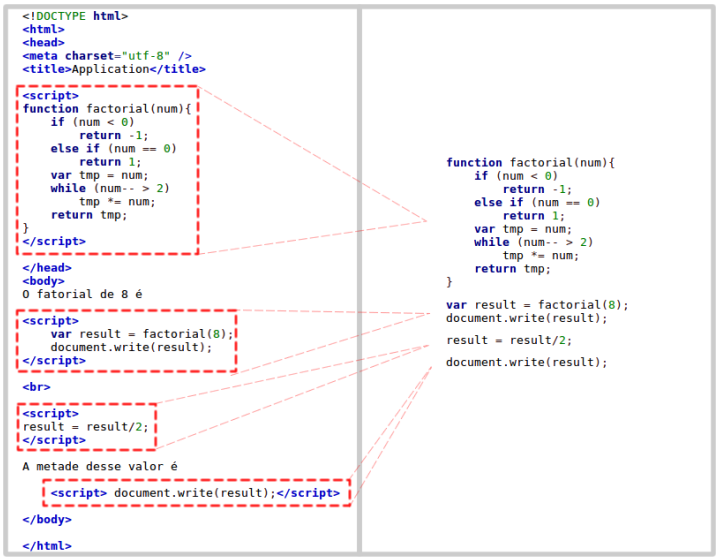

Figura 2. Etapas da fase de identificação e extração de códigos JavaScript

3.2. Criação da Árvore Sintática Abstrata

As ASTs são usadas intensivamente pelos compiladores durante a análise semântica de um código-fonte, onde o compilador verifica o uso correto dos elementos do programa. Neste trabalho, nós usamos as ASTs para fornecer uma representação de todas as entidades, objetos, propriedades, funções, estruturas e relações dos blocos de código extraídos na fase anterior.

O mecanismo de criação da AST do *TaintJSec* possibilita a inserção de novas informações na árvore por meio de processamento subsequente. Por exemplo, é possível adicionar o número da posição exata de um trecho de código *JavaScript* responsável pela criação de um nó específico da árvore, ou adicionar qualquer atributo de valor relevante com o propósito de enriquecer a AST com informações úteis que facilitem a análise do fluxo da informação. Assim, a árvore sintática gerada pelo *TaintJSec* tem uma estrutura flexível que permite a inserção de novos atributos e a alteração dos dados presentes nos nós a qualquer momento da análise.

3.3. Análise do Fluxo da Informação

A terceira fase da abordagem realiza a análise do fluxo de execução do código *JavaScript* para identificar possíveis vazamentos de informações sensíveis. Para que a análise seja realizada são necessários dois artefatos de entrada: o código *JavaScript* e a sua respectiva representação em AST, ambos são artefatos de saída das fases anteriores. O código *JavaScript* serve de apoio para a análise, onde são feitas consultas com o propósito de obter o trecho real do código gerador da AST.

O analisador visita todas as entidades, objetos, propriedades, funções e blocos de código do programa e armazena suas estruturas e relações. Simultaneamente, nós instrumentamos o código para monitorar todos os registradores, estruturas de dados que carregam alguma informação definida como sensível. As seções seguintes fornecem detalhes desse processamento.

3.3.1. Estrutura dos Registradores

Um registrador de objeto no *TaintJSec* é uma estrutura de dados que contém informações relevantes, tais como o identificador (nome de variável ou função), o valor, o tipo e os atributos do objeto. Os registradores são utilizados pelo *TaintJSec* para armazenar os dados obtidos durante a análise do fluxo de código. Esses registradores diferenciam o escopo ao qual pertencem, para que assim não ocorra ambiguidade entre os dados armazenados.

O processo utilizado para armazenar e atualizar os objetos de um determinado escopo na medida em que a análise consome os níveis da AST é apresentado no Algoritmo 1. Caso o escopo já exista, então o objeto é inserido no conjunto de objetos do escopo (linha 7), de modo que nesse conjunto de objetos não é permitido a existência de dois registradores com o mesmo identificador. Logo, a inserção de um objeto com um identificador já inserido anteriormente sobrescreve o objeto mais antigo.

Algoritmo 1. Armazenamento e atualização de objetos em registradores

```
   Entrada: (S, u, k)
   Saída: Conjunto de escopos atualizado
   /* onde S é o conjunto de todos os escopos conhecidos,
      */
   /* u é um escopo específico, e                          */
   /* k é um objeto do escopo u                            */
 1 início
 2 |  i ← k.identificador
 3 |  se u ∉ S então
 4 |  |  S ← S ∪ {u}
 5 |  |  σ(S, u) ← { i : k }
 6 |  senão
 7 |  |  σ(S, u, i) ← k
 8 |  fim
 9 fim
10 retorna S
```

A estrutura de registradores deve ser flexível o suficiente para permitir que novas informações sejam inseridas em todos os registradores ou somente em um registrador específico. Essa característica é útil para detectar se a função *eval* contém informação sensível em seus argumentos, bem como para propagar um novo atributo inserido pela propriedade *prototype* a todas as instâncias de uma classe.

Para evitar ambiguidade nas informações armazenadas, cada escopo possui sua própria estrutura de registradores. De tal maneira que dois escopos com identificadores **1.1** e **1.2**, por exemplo, são escopos adjacentes e também subescopos de um terceiro que possui identificador igual a **1**. Para exemplificar o funcionamento dessas estruturas, o código $\boldsymbol{answer = 6 * 7}$ é utilizado como exemplo na Figura 3.

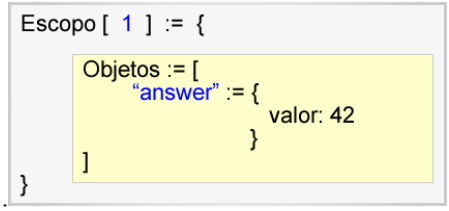

Figura 3. Exemplo da estrutura gerada para o código $\boldsymbol{answer = 6 * 7}$

3.3.2. Marcação de Registradores

Paralelamente a análise da AST, os registradores que estiverem portando informação sensível são marcados (*taint tag*), para que o algoritmo do *TaintJSec* possa identificar possíveis vazamentos de dados. Neste caso, o *TaintJsec* insere um campo *taint tag* na estrutura dos registradores para sinalizar que o objeto analisado está manipulando uma informação sensível. Por exemplo, considerando o registrador exibido na Figura 3, o campo *taint tag* seria marcado com valor *falso*, indicando que esse objeto não contém uma informação sensível.

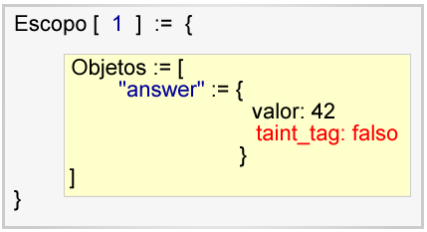

Figura 4. Exemplo da taint tag inserida na estrutura de um registrador

3.3.3. Análise de Marcação (Taint Analysis)

Como citado anteriormente, a análise do fluxo da informação consiste em marcar os dados que são identificados como dados sensíveis e observar por onde esses dados marcados (*taint data*) são propagados. Assim, a técnica consegue identificar o momento que um *taint data* propaga-se de um objeto para outro (*taint propagation*).

Neste trabalho, todos os métodos que fornecem informação sensível (*taint sources*) são identificados e armazenados em uma lista para que sirvam de consulta sempre que a análise do fluxo de execução encontrar uma operação de chamada de função ou método. Tal consulta visa identificar se o objeto a ser executado é ou não um *taint source*. Desta forma, todo objeto que armazenar informações provenientes de um *taint source*, tem o valor da *taint tag* alterado para *verdadeiro*, sinalizando que a partir daquele momento o objeto está manipulando uma informação sensível.

Da mesma forma, todo método ou função presente na linguagem *JavaScript* que possibilite o envio de informação sensível para um destino que esteja fora do domínio da aplicação é considerado como *taint sink*. Neste caso, um vazamento de dados é caracterizado sempre que um método enviar qualquer informação marcada como sensível para fora da aplicação, como mostrado no Algoritmo 2.

Algoritmo 2. Detecção de vazamento de informação

Entrada: (Θ, η, α)
Saída: O valor booleano que indica a detecção do vazamento de informação
/* onde Θ é o conjunto de todos os taint sinks, e */
/* η é um elemento JavaScript, e */
/* α é um registrador */
1 **início**
2 | $vazamento \leftarrow 0$
3 | **se** $TSINK(\Theta, \eta)$ **então**
4 | | $vazamento \leftarrow \alpha.taint\_tag$
5 | **fim**
6 **fim**
7 **retorna** $vazamento$

3.3.3.1. Propagação da Taint Tag

Todo valor resultante de uma operação que envolve *taint data* leva consigo a *taint tag* positiva. Esse valor é repassado a outros objetos de diferentes maneiras, podendo ser por uma simples atribuição de valor entre objetos ou por meio do envio a uma função em forma de parâmetro, por exemplo.

Desse modo, faz-se necessário criar um conjunto de regras de propagação da *taint tag* em meio às diversas operações no fluxo de execução do código *JavaScript*. Essas regras dividem-se em propagação por atribuição, propagação por expressão, propagação por passagem de parâmetro e propagação por execução de função nativa. Cada uma dessas regras é executada de acordo com o tipo de operação encontrada durante a análise da árvore sintática abstrata.

A propagação por atribuição é realizada quando uma variável recebe o valor diretamente de algum *taint source* ou de um *taint data*. A propagação por expressão é realizada quando um ou mais elementos de uma expressão possuem *taint tag* positiva, fazendo com que o valor resultante da expressão também o tenha. A propagação por passagem de parâmetro ocorre quando pelo menos um dos argumentos enviados como parâmetro a uma função não-nativa possui *taint tag* positiva. Por fim, a propagação por função nativa ocorre quando pelo menos um dos argumentos da função é um *taint data*.

Propagação na Função Eval (Caso especial de propagação por função nativa)

A função *eval* é uma função nativa utilizada para criar novos trechos de código em tempo de execução. Em geral, os trabalhos que utilizam a análise estática em código *JavaScript* não oferecem suporte à função *eval* em razão das dificuldades encontradas durante a análise de sua execução. Como resultado, a função *eval* torna-se um problema por ser bastante utilizada em aplicações maliciosas que praticam o vazamento de dados sensíveis.

Para resolver esse problema, este trabalho propõe um novo conceito chamado *taint position*.

Definição 1: *Taint Position é o nome dado ao número que identifica a posição de um taint data contido em um valor literal*.

Um *taint position* é criado após uma operação de concatenação entre dois objetos em que pelo menos um deles é um *taint data*. O resultado dessa operação é armazenado no registrador de objetos junto com a *taint tag* positiva e um conjunto de *taint position* que indica a posição exata de um ou mais *taint data* existentes no valor literal.

O conjunto de *taint position* é o resultado da união do conjunto de *taint position* do elemento da esquerda com o da direita. Essa operação de união de conjuntos obedece às regras apresentadas abaixo:

(R1) O elemento da operação que não for *taint data* fornece um conjunto de *taint position* vazio;

(R2) O *taint data* do lado esquerdo da operação fornece um conjunto unitário contendo o *taint position* 1, quando este não possuir um conjunto de *taint position* no registrador;

(R3) O *taint data* do lado esquerdo da operação fornece o conjunto de *taint position* de seu registrador, caso possua;

(R4) O *taint data* do lado direito da operação fornece um conjunto unitário contendo o número resultante da operação $1 + comprimento\ do\ elemento\ da\ esquerda$, quando não possuir um conjunto de *taint position* em seu registrador;

(R5) O *taint data* do lado direito da operação que possui um conjunto X de *taint position* em seu registrador, fornece o conjunto imagem $I(f)$, onde $f : X \rightarrow Y$ e $f(x) = x + comprimento\ do\ elemento\ da\ esquerda$.

O Algoritmo 3 apresenta como as cinco regras acima são utilizadas para fornecer um conjunto de *taint position* resultante de uma operação de concatenação entre dois objetos.

Algoritmo 3: Criação do conjunto de taint positions resultante de uma operação de concatenação entre dois objetos

```
   Entrada: (Reg^esq, Reg^dir)
   Saída: Conjunto de taint position
   /* onde Reg^esq é o registrador do objeto da esquerda, e */
   /* Reg^dir é o registrador do objeto da direita          */
 1 início
 2 |  conj^esq ← { } ;                                        // (R1)
 3 |  conj^dir ← { } ;                                        // (R1)
 4 |
 5 |  se Reg^esq.taint_flag é positivo então
 6 |  |  se Reg^esq.taint_position existe então
 7 |  |  |  conj^esq ← Reg^esq.taint_position ;               // (R3)
 8 |  |  senão
 9 |  |  |  conj^esq ← { 1 } ;                               // (R2)
10 |  |  fim
11 |  fim
12 |  se Reg^dir.taint_flag é positivo então
13 |  |  se Reg^dir.taint_position existe então
14 |  |  |  para cada i ∈ Reg^dir.taint_position faça
15 |  |  |  |  conj^dir ← conj^dir ∪ { i + TAMANHO(Reg^esq.valor) } ;  // (R5)
16 |  |  |  fim
17 |  |  senão
18 |  |  |  conj^dir ← { 1 + TAMANHO(Reg^esq.valor) } ;      // (R4)
19 |  |  fim
20 |  fim
21 fim
22 S ← conj^esq ∪ conj^dir
23 retorna S
```

A Tabela 1 exemplifica como ocorre a criação de um *taint position*, onde é possível observar que o código **var B = "texto" + A;** resulta em um *taint data* contendo um conjunto de *taint position* com apenas um elemento, sendo ele o *taint position* 6. Isso ocorre porque a variável **A** é marcada (*taint data*), cujo valor inicia a partir do sexto caractere da literal resultante **"texto127.0.0.1"** da operação de concatenação.

Tabela 1. Exemplo de criação de taint position

Código	Registradores
var A = getIP();	R_A = { valor: "127.0.0.1", taint: 1 }
var B = "texto" + A;	R_B = { valor: "texto127.0.0.1", taint: 1, taint_position: [6] }
var C = A + B;	R_C = { valor: "127.0.0.1texto127.0.0.1", taint: 1, taint_position: [1, 15] }

O código **var C = A + B;** da Tabela 1, apresenta uma operação de concatenação entre dois *taint data* (**A** e **B**) que resulta em um *taint data* com novos *taint positions*. Neste caso, o conjunto de *taint position* é obtido a partir da união do conjunto de *taint*

position do elemento da esquerda da operação (variável **A**) com o conjunto de *taint position* do elemento da direita da operação (variável **B**).

O elemento da esquerda obedece a R2 e fornece o conjunto {1}, enquanto o elemento da direita, cujo registrador contém o *taint position* 6, obedece a R5 e fornece o conjunto {15}, pois $\mathbf{6 + 9 = 15}$, onde o número 9 equivale ao comprimento do valor de **A**. Como resultado, o conjunto de *taint position* após a operação de concatenação é {1, 15}.

Assim, a utilização de *taint position* como solução para o problema da função *eval* mostra-se eficaz, pois é possível propagar essa informação para o escopo da função *eval* e informar ao analisador a posição dos *taint data* presentes no argumento da função.

No entanto, a obtenção do conjunto de *taint position* deve ocorrer durante o processo de reescrita do parâmetro de entrada da função *eval*. Esse processo consiste em reescrever o argumento enviado à função *eval*, da esquerda para a direita, substituindo todos os identificadores de variáveis pelos seus respectivos valores. Para cada variável encontrada, seu registrador é consultado para checar o valor da *taint tag*. Quando um *taint data* é encontrado, o conjunto de *taint position* do elemento da direita da concatenação é obtido conforme as regras estabelecidas para a criação do conjunto de *taint position*, e serve de consulta durante a análise da execução da função *eval* (Figura 5). Esse processo ocorre até que não existam mais variáveis a serem substituídas, sobrando apenas uma única *string*.

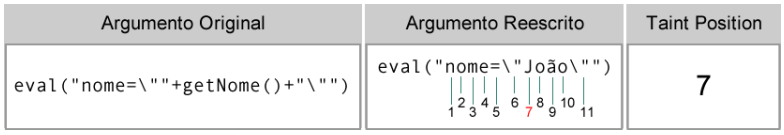

Figura 5. Exemplo do processo de reescrita do argumento da função eval e identificação dos taint positions

Com o argumento reescrito e os *taint positions* conhecidos, é criada a árvore sintática abstrata do argumento, contendo o intervalo do código-fonte que originou cada nó da árvore. Todo nó da AST que contém um intervalo de código-fonte onde pelo menos um dos elementos do conjunto de *taint position* está inserido resulta em um *taint data*, cuja *tag* é propagado para os níveis superiores da árvore.

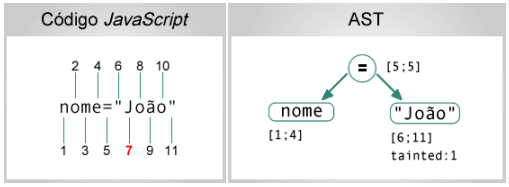

Figura 6. Exemplo de AST com os atributos de intervalo de código-fonte e a taint tag atribuída devido a existência de um taint position no intervalo

A Figura 6 apresenta a AST do argumento reescrito da Figura 5 cuja operação de atribuição resulta em um *taint data*, pois o *taint position* 7 está presente no intervalo de 6 a 11, atributo do nó à direita da árvore.

4. Experimentos e Resultados

Nesta seção são apresentados os testes realizados para validar a eficácia do *TaintJSec* na detecção do vazamento de informação sensível. Este capítulo inicia com a apresentação do protocolo experimental, onde são descritos os passos realizados para a criação dos experimentos. Em seguida, são apresentados os testes e seus respectivos resultados.

4.1. Protocolo Experimental

Para montar a base de testes foram obtidos códigos *JavaScript* de quatro fontes distintas, sendo elas: códigos aleatórios gerados pela ferramenta *Eslump*; códigos obtidos na base de testes do *framework* SAFE; códigos da base de testes do *JSPrime*; e, códigos extraídos de artigos. Foi realizada uma tarefa de identificação e exclusão de códigos similares. Essa tarefa foi necessária para evitar testes desnecessários, onde somente nomes de variáveis ou valores eram alterados em um código que utiliza a mesma lógica ou o mesmo recurso da linguagem. Todos os códigos da base tiveram seus respectivos códigos similares excluídos.

Após a tarefa de exclusão de similares sobraram 309 casos de testes, os quais foram agrupados em 14 conjuntos de testes de acordo com a operação *JavaScript* que executavam, como mostra a Tabela 2.

Tabela 2. Conjuntos de testes realizados para testar a propagação da taint tag

Conjuntos	Testes	Número de Operações
1	Atribuição Encadeada	3
2	Atribuição Simples	37
3	Condicional	22
4	Função Não-Nativa	50
5	Função Nativa	21
6	Função Nativa (eval)	10
7	Prototype	56
8	Laço de Repetição (do while)	11
9	Laço de Repetição (for)	9
10	Laço de Repetição (for in)	6
11	Laço de Repetição (while)	8
12	Switch	20
13	Try-Catch	26
14	Vetor	30

Os 14 conjuntos de testes foram utilizados para testar a qualidade computacional do analisador estático e a corretude da propagação da *taint tag*. Para validar os resultados obtidos pelo *TaintJSec* em operações de expressões matemáticas, execução de funções nativas ou qualquer interação entre os objetos do código, os testes foram comparados com os resultados do *SpiderMonkey*.

Foram organizadas três baterias de testes:

(1) Testes de Propagação - para validar a correta propagação da *taint tag* nos diferentes tipos de operações *JavaScript*, tais como as operações de: atribuição; condicionais; laços de repetição; *switch*; *try-catch*; execução de funções; e, inserção de novos atributos em instâncias de objetos, por meio do *prototype*.

(2) Testes com a Função *eval* - para validar a propagação da *taint tag* utilizando *taint position*. Nessa bateria de teste foi utilizado o conjunto de testes com 10 casos distintos de execução da função *eval* contendo operações como: concatenação de *taint data*; inclusão de objetos portadores de *taint data*; chamadas recursivas; e, operações encadeadas. Além dos casos de teste, também foi desenvolvida uma aplicação maliciosa, cujo código *JavaScript* apresenta uma combinação da função *eval* com técnicas de ofuscação para praticar o vazamento de informação.

(3) Testes em Códigos Ofuscados - para avaliar a capacidade do *TaintJSec* em detectar vazamento de informação em códigos ofuscados por ferramentas específicas para tal finalidade. Foram realizados testes utilizando códigos ofuscados a partir de ferramentas como *JSCompress*, *Aaencode*, *JSFuck*, *JSCrambler* e *Packer*.

A seguir, a Figura 7 apresenta uma visão geral do processo de criação dos experimentos utilizados para validar a proposta deste trabalho.

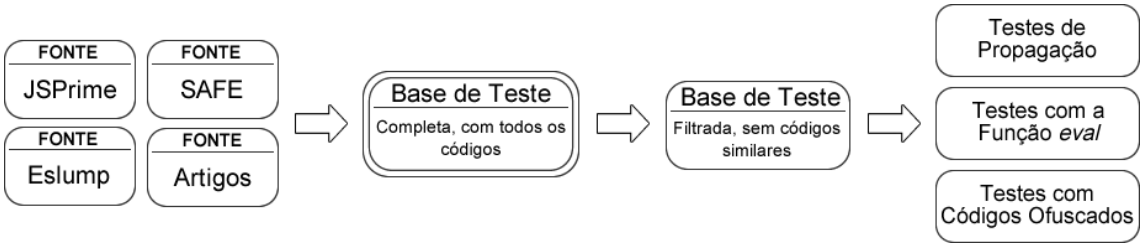

Figura 7. Visão geral do protocolo experimental

4.2. Resultados

A Tabela 3 apresenta os resultados obtidos utilizando os 14 conjuntos de testes, onde é possível observar que a acurácia da abordagem proposta é superior às ferramentas *JPrime, JSpwn, ScanJS, JSLint, Jalangi e jsTaint.*

Tabela 3. Comparação entre os trabalhos relacionados e a abordagem proposta

Trabalhos	Conjuntos de Testes														Total	%
	1	2	3	4	5	6	7	8	9	10	11	12	13	14		
JSPrime	3	37	22	50	21	6	56	11	9	6	8	20	26	30	305	98
JSpwn	3	37	22	50	15	6	0	11	9	6	8	20	26	30	243	78
ScanJS	3	37	22	50	15	8	0	11	9	6	8	20	26	30	245	79
JSLint	3	37	22	42	8	0	0	0	0	0	0	0	14	30	156	50
Jalangi	3	37	22	50	21	8	56	11	9	6	8	20	26	30	307	99
jsTaint	3	37	22	25	0	0	0	11	9	6	8	20	26	30	197	63
TaintJSec	3	37	22	50	21	10	56	11	9	6	8	20	26	30	309	100

O comportamento do *TaintJSec* durante a análise da função *eval* foi satisfatório, atingindo uma acurácia de 100% para os testes realizados para avaliar o método de detecção de propagação da *taint tag* utilizando *taint positions*.

Os resultados dos testes em códigos ofuscados mostraram que a ofuscação não foi impedimento para que o vazamento de informação fosse detectado pela abordagem. Notou-se que as ferramentas de ofuscação fazem uso da criação de código em tempo de execução, e por esse motivo o *TaintJSec* foi preciso na detecção do vazamento de

informação, enquanto outros trabalhos falharam por não oferecer um analisador que suportasse tal característica da linguagem *JavaScript*.

5. Conclusão e Trabalhos Futuros

Este trabalho apresentou o *TaintJSec*, uma abordagem de análise estática, para a detecção de vazamento de informação sensível em código *JavaScript*. Diferente das abordagens existentes, o *TaintJSec* é capaz de verificar código *JavaScript* explícitos e implícitos. Mais ainda, o *TaintJSec* consegue analisar a propagação de um dado sensível na execução da função *eval*, a qual é bastante utilizada em técnicas de ofuscação de código. Além de conseguir identificar o vazamento de informação sensível em códigos ofuscados por ferramentas criadas especificamente para tal finalidade.

A avaliação do *TaintJSec* utilizando 14 conjuntos de testes mostraram que a acurácia da abordagem proposta é superior às ferramentas *JPrime, ScanJS, JSpwn, KSLint, Jalangi e jsTaint.* Entretanto, apesar do *TaintJSec* ter obtido bons resultados, há muitas questões a serem melhoradas, como a análise do fluxo durante a chamada dos *Event Listeners* e o suporte a *callbacks* como o objeto *new Proxy*, por exemplo.

Neste trabalho foram identificadas as seguintes possibilidades de trabalhos futuros: A inclusão completa do DOM na análise para reconhecer funções nativas dos navegadores. Estender o *taint propagation* pelos objetos do HTML para permitir a marcação de *tags* HTML que possuam dados sensíveis. A criação de um controle de *taint tags* para os objetos do tipo *array*, pois atualmente existe somente uma *taint tag* para o objeto inteiro, de tal maneira que esse controle poderia observar a propagação da *taint tag* por posição do objeto, diminuindo o número de falsos positivos.

Agradecimentos

Este trabalho foi apoiado parcialmente pelo SIDIA – Samsung Instituto de Desenvolvimento para a Informática da Amazônia.

Referências

Electron. (2018). *Build Cross Platform Desktop Apps with JavaScript, HTML and CSS.* Retrieved from https://electronjs.org/

Fard, A. M., & Mesbah, A. (2013, 9). JSNOSE: Detecting JavaScript Code Smells. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, (pp. 116-125). doi:10.1109/SCAM.2013.6648192

Fard, A. M., & Mesbah, A. (2017, 3). JavaScript: The (Un)Covered Parts. pp. 230-240. doi:10.1109/ICST.2017.28

Gibler, C., Crussell, J., Erickson, J., & Chen, H. (2012). AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. *Proceedings of the 5th International Conference on Trust and Trustworthy Computing* (pp. 291-307). Berlin: Springer-Verlag. doi:10.1007/978-3-642-30921-2_17

Google, L. L. (2018). *What Are Extensions?* Retrieved from https://developer.chrome.com/extensions

Hsiao, S. W., Hung, S.-H., Chien, R., & Yeh, C. W. (2014). PasDroid: Real-Time Security Enhancement for Android. *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 229-235.

Kashyap, V., Dewey, K., Kuefner, E. A., Wagner, J., Gibbons, K., Sarracino, J., . . . Hardekopf, B. (2014). JSAI: A Static Analysis Platform for JavaScript. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 121-132). New York, NY, USA: ACM. doi:10.1145/2635868.2635904

Kuzuno, H., & Tonami, S. (2013, 4). Signature Generation for Sensitive Information Leakage in Android Applications. *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, (pp. 112-119). doi:10.1109/ICDEW.2013.6547438

Monteiro, D., Patnaik, N. D., & Theriault, P. (2013). JSpwn - JavaScript Static Code Analysis.

Mozilla. (2018). *WebExtensions | MDN*. Retrieved from https://developer.mozilla.org/pt-BR/Add-ons/WebExtensions

Node.js. (2018). *Linux Foundation*. Retrieved from https://nodejs.org

Ocariza, F., Bajaj, K., Pattabiraman, K., & Mesbah, A. (2013, 10). An Empirical Study of Client-Side JavaScript Bugs. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, (pp. 55-64). doi:10.1109/ESEM.2013.18

Patnaik, N. D., & Sahoo, S. S. (2015). JSPrime - A JavaScript Static Security Analysis Tool.

Peneti, S., & Rani, B. P. (2016, 2). Data leakage Prevention System with Time Stamp. *2016 International Conference on Information Communication and Embedded Systems (ICICES)*, (pp. 1-4). doi:10.1109/ICICES.2016.7518934

Pinto, B. S., Boeira, F. C., Minatel, P., Pires, P. C., Souza, I., Silva, A., & Shin, J. (2016). Mobile Data Leakage Prevention using Packet Inspection Approach.

Theriault, P. (2013). ScanJS - Static Analysis Tool for JavaScript Code.

Tizen. (2018). *An Open Source, Standards-based Software Platform for Multiple Device Categories.* Retrieved from https://www.tizen.org/

Wang, D., Jin, G., He, J., Jiang, X., & Xie, Z. (2014). A Grey List-Based Privacy Protection for Android. *JSW, 9*, 1525-1531.

WinJS. (2018). *WinJS - A Windows Library for JavaScript*. Retrieved from http://www.buildwinjs.com/