Um sistema para análise e detecção de ataques ao navegador *Web*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Vitor Monte Afonso e aprovada pela Banca Examinadora.

Campinas, 13 de outubro de 2011.

Paulo Lício de Geus (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Substitua pela ficha catalográfica

(Esta página deve ser o verso da página anterior mesmo no caso em que não se imprime frente e verso, i.é., até 100 páginas.)

Substitua pela folha com as assinaturas da banca

Instituto de Computação
Universidade Estadual de Campinas

Um sistema para análise e detecção de ataques ao navegador *Web*

Vitor Monte Afonso

Outubro de 2011

Banca Examinadora:

- Paulo Lício de Geus (Orientador)
- Julio Cesar López Hernández
- Adriano Mauro Cansian
- Eliane Martins (suplente)
- Carlos Alberto Maziero (suplente)

Resumo

Páginas *Web* com conteúdo malicioso são uma das grandes ameaças à segurança de sistemas atualmente. Elas são a principal forma utilizada por atacantes para instalar programas maliciosos (*malware*) no sistema operacional dos usuários. Para desenvolver mecanismos de proteção contra essas páginas, elas precisam ser estudadas e entendidas profundamente. Existem diversos sistemas de análise que são capazes de analisar páginas *Web*, prover informações sobre elas e classificá-las como maliciosas ou benignas. Entretanto, estes sistemas possuem diversas limitações em relação ao tipo de código que pode ser analisado e aos tipos de ataque que podem ser detectados.

Para suprir tal deficiência nos sistemas de análise de páginas *Web* maliciosas foi desenvolvido um sistema, chamado de BroAD (*Browser Attacks Detection*), que faz a análise destas páginas de forma dinâmica, monitorando tanto as chamadas de sistemas realizadas pelo navegador enquanto as processa, quanto as ações realizadas pelo código JavaScript contido na página. A detecção dos comportamentos maliciosos é feita em quatro etapas, utilizando técnicas de aprendizado de máquina e assinaturas. Estas etapas incluem a detecção de *shellcodes*, a detecção de anomalias no comportamento do JavaScript e a análise de chamadas de sistema e assinaturas de código JavaScript.

Foram realizados testes que demonstram que o sistema desenvolvido possui taxas de detecção superiores aos sistemas do estado-da-arte de análise de páginas *Web* maliciosas e ainda provê mais informações a respeito delas, levando a um entendimento melhor das amostras. Além disso, são apresentados códigos que podem detectar e evadir facilmente a análise de parte desses sistemas usados na comparação, demonstrando a fragilidade deles.

Abstract

Malicious Web applications are a significant threat to computer security today. They are the main way through which attackers manage to install malware on end-user systems. In order to develop protection mechanisms to these threats, the attacks themselves need to be deeply studied and understood. Several analysis systems exist to analyze Web pages, provide information about them and classify them as malicious or benign. However, these systems are limited regarding the type of attacks that can be detected and the programming languages that can be analyzed.

In order to fill this gap, a system, called BroAD (Browser Attacks Detection), that is capable of analyzing malicious Web pages, was developed. It monitors both system calls and JavaScript actions and the detection of the malicious behavior is performed in four steps, by the use of machine learning techniques and signatures. These steps include the detection of shellcodes, the anomaly detection of the JavaScript behavior and the analysis of system calls and JavaScript signatures.

The results of the performed tests show that the developed system has better detection rates than the state-of-the-art systems in malicious Web pages analysis and also provides more information about these pages, giving a better understanding about their behavior. Besides, codes that can be used to easily detect and evade the analysis of part of those systems are presented, showing their fragility.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Nos últimos anos, tem havido um grande aumento e progresso nos serviços oferecidos através de sistemas *Web*. Atividades como compras e operações bancárias, que antes requeriam o deslocamento físico dos usuários, estão sendo realizadas remotamente. Com isso, as tecnologias que são executadas no lado do cliente (i.e., o usuário) em navegadores *Web* evoluíram bastante, tornando-se muito comuns e necessárias para proporcionar uma melhor interação entre o usuário e os sistemas remotos, ao mesmo tempo em que tiveram sua complexidade aumentada. Entretanto, tais mudanças também fizeram com que os usuários se tornassem alvos mais interessantes, dado que, através do comprometimento da máquina do usuário o atacante pode, por exemplo, adquirir seus dados bancários e realizar operações forjando sua identidade. Mesmo que o volume financeiro movimentado a partir de um usuário seja muito menor do que aquele possível através do comprometimento do servidor de um banco, a dificuldade e risco deste tipo de ataque e a imensa quantidade de possíveis usuários "alvos" faz com que a relação custo/benefício torne favorável os ataques a usuários finais, de forma automatizada. Devido a este cenário, a incidência dos ataques que têm como objetivo atingir os usuários através do abuso de seus navegadores *Web* aumentou [21], criando um grande problema.

Para comprometer as vítimas, um atacante precisa executar certos códigos em seus navegadores *Web*. Isto é alcançado através da hospedagem de códigos maliciosos em alguma página na *Web*, seguida de tentativas de levar o usuário a carregar esse código, acessando a página maliciosa. Para completar esta etapa, os atacantes fazem uso de técnicas de engenharia social, como *phishing*, e da infecção de páginas legítimas[1]. Existem muitos *sites* hospedando esse tipo de código sem que seus proprietários saibam. Isto se deve à dificuldade em se criar sistemas *Web* com segurança, dado que muitas vezes os desenvolvedores não possuem os conhecimentos necessários de desenvolvimento seguro, ou a

[1]No contexto desse trabalho, uma página legítima é aquela que não tem por objetivo hospedar códigos que podem causar mal aos usuários.

segurança é deixada de lado por restrições de tempo e orçamento. Além disso, caso um atacante hospede conteúdo malicioso em um *site* popular, seu código será executado por uma quantidade grande de usuários, como nos casos dos *worms* Samy[2] e Yamanner [8]. Uma vez que o navegador *Web* da vítima carrega o conteúdo malicioso, o código carregado extrai alguma informação pessoal da vítima ou abusa de alguma vulnerabilidade do navegador ou de um de seus componentes, resultando na instalação de algum programa malicioso—*malware*—no sistema.

O código malicioso executado no navegador da vítima, chamado alternadamente a partir deste ponto de *Web malware* e páginas *Web* maliciosas, é comumente desenvolvido na linguagem JavaScript e é usado para, entre outras atividades, roubar informações pessoais do usuário, tomar controle do navegador, abusar de alguma vulnerabilidade e instalar *malware* no sistema. Estes *malware* são comumente utilizados para enviar mensagens de *spam*, atacar outros sistemas através da rede ou roubar informações do usuário, como senhas e dados bancários. Os *Web malware* estão constantemente evoluindo, forçando as empresas que desenvolvem soluções de defesa a estudar e entender profundamente as técnicas empregadas e as vulnerabilidades que sofrem abuso. Um tipo de defesa constantemente utilizada são as *blacklists*, que são listas de URLs (*Universal Resorurce Location*) que possuem código malicioso. Os navegadores consultam estas listas e verificam, enquanto o usuário navega, se as páginas acessadas fazem parte delas, bloqueando o acesso em casos positivos. Como as páginas maliciosas mudam constantemente, as empresas que criam e mantêm essas listas (como o serviço *safe browsing* do Google[3] e o *smart screen filter* da Microsoft[4]) precisam analisar as páginas para identificar quais delas contêm códigos maliciosos e devem ser inseridas nas listas e quais necessitam ser retiradas das listas.

Diversas abordagens foram propostas para analisar páginas maliciosas. Em [9], os autores apresentam um sistema que faz análise de código JavaScript e utiliza aprendizado de máquina para detectar páginas maliciosas, porém esse sistema está limitado a ataques que utilizam código JavaScript. Em [38] é apresentado outro sistema de análise que pode tratar, além de JavaScript, código VBScript e detecta, através da emulação de vulnerabilidades, tentativas de abuso do navegador. Já em [43] é apresentado um sistema que monitora as chamadas de sistema feitas pelo navegador e realiza a detecção de comprometimento do navegador através dessas chamadas de sistema. Apesar de serem os sistemas de análise mais populares, estes possuem falhas, ou por serem capazes de analisar apenas parte dos ataques existentes ou por serem capazes de detectar apenas ataques bem sucedidos.

Para estender a capacidade das abordagens de análise de *Web malware* existentes, foi

[2]http://namb.la/popular/
[3]http://code.google.com/apis/safebrowsing/
[4]http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter

desenvolvido um sistema que alia o monitoramento de código JavaScript com a captura de chamadas de sistema efetuadas pelo navegador e uma metodologia de detecção em quatro etapas. O sistema proposto provê informações acerca do comportamento do código analisado e das modificações efetuadas no sistema após sua execução, e classifica a amostra como maliciosa ou benigna, baseado nas informações coletadas e em um treinamento prévio com milhares de exemplares de ambos os tipos. Para fazer o monitoramento do JavaScript é utilizada uma biblioteca de Windows, a qual é carregada na memória do navegador Internet Explorer, e um *driver* de *kernel*, o qual captura as chamadas de sistema. As quatro etapas da detecção incluem a verificação de assinaturas das ações feitas em JavaScript, de *shellcodes*, de chamadas de sistema e a detecção de anomalia no código JavaScript. Para demonstrar a efetividade do sistema desenvolvido foram feitos testes com 8691 amostras, incluindo benignas e maliciosas, e o resultado do sistema foi comparado com o resultado de três sistemas conhecidos e publicamente disponíveis. Estes testes demonstram que a taxa de detecção do sistema desenvolvido é consideravelmente melhor do que a dos outros sistemas. De forma resumida, as principais contribuições deste trabalho são:

- É proposta uma nova arquitetura que unifica parte das abordagens de detecção existentes, estendendo a análise tradicional para a inspeção de códigos seguida do monitoramento das modificações no sistema;
- Foi desenvolvida uma metodologia de detecção que permite que o sistema detecte tanto ataques de *drive-by download* quanto ataques de roubo de informações, sem estar inteiramente limitado à linguagem JavaScript, preenchendo uma lacuna na área;
- Foi desenvolvido um protótipo e este foi testado com páginas maliciosas, e o resultado foi comparado com as ferramentas do estado-da-arte existentes, demonstrando que a abordagem desenvolvida produz melhores taxas de detecção que as existentes ao mesmo tempo em que provê mais informações sobre o comportamento do código;
- São apresentados códigos desenvolvidos que podem detectar que os *Web malware* estão sendo analisados por ferramentas do estado-da-arte, permitindo-os ocultar seu comportamento malicioso de forma a evadir a análise e subverter as ferramentas. O sistema desenvolvido é imune às técnicas de detecção e evasão empregadas nesses códigos, podendo analisá-los e obter o comportamento que seria executado no sistema da vítima.

O restante desta dissertação está organizado como segue:

Capítulo 2: são descritas as formas empregadas pelos adversários para realizar ataques contra os usuários, bem como os principais tipos de ataque realizados para roubar informações dos usuários e instalar *malware* em seus sistemas. Além disso, são mostradas as técnicas utilizadas para dificultar a análise dos códigos maliciosos e impedir que grupos de pesquisa em segurança e empresas que desenvolvem sistemas de proteção tenham acesso a esses, dificultando a criação de contra-medidas. Também são apresentados códigos que podem evadir a análise feita por sistemas atuais, escondendo o comportamento malicioso.

Capítulo 3: neste capítulo, são descritas as principais técnicas utilizadas para analisar os exemplares de *Web malware*, separadas pelo uso ou não de um emulador de navegador, e as abordagens existentes para realizar a proteção dos usuários, incluindo sistemas que atuam apenas no cliente, que atuam tanto no cliente como no servidor e que atuam apenas no servidor.

Capítulo 4: neste capítulo é descrito o sistema de análise desenvolvido, o papel de cada um de seus componentes, as interações entre eles, a configuração do ambiente de análise, como se dá seu processo, as informações providas ao final deste e as técnicas de detecção de comportamento malicioso utilizadas.

Capítulo 5: são apresentados os testes realizados para comparar o sistema desenvolvido com três dos sistemas existentes (e disponíveis publicamente) mais populares. São discutidos também os resultados de tal comparação e é feita uma análise destes, mostrando a efetividade do sistema desenvolvido.

Capítulo 6: neste capítulo são apresentadas as conclusões e considerações finais, bem como os trabalhos futuros que podem ser derivados deste.

Capítulo 2

Ataques

Neste capítulo são descritos os tipos de ataque via *Web* realizados contra os usuários no lado do cliente, o modo como estes ataques são realizados e quais as técnicas usadas por atacantes para impedir que seus códigos sejam analisados por pesquisadores de segurança e empresas que desenvolvem mecanismos de defesa (por exemplo, sistemas anti-vírus). Além disso, são apresentados códigos que podem evadir a análise de certos sistemas.

Para alcançar os usuários, os atacantes precisam hospedar seus códigos maliciosos em algum *site*, a fim de que este seja acessado posteriormente. Esta etapa muitas vezes envolve o comprometimento de aplicações legítimas, como por exemplo, a aplicação que opera como servidor *Web*. Sendo assim, neste capítulo também são detalhados os ataques mais comumente utilizados para inserir código malicioso nestas aplicações.

2.1 Segurança de *Software*

A falta de investimentos em segurança, ou mesmo a baixa prioridade dada ao assunto por parte das empresas que desenvolvem *software*, aliada à não inclusão deste tema em cursos de desenvolvimento de *software* fizeram com que se chegasse ao estado problemático atual, no qual muitas empresas acabam sendo vítimas de ataques [28] e podem ter seus dados roubados, alterados ou destruídos.

Os ataques em que os navegadores *Web* dos usuários são abusados e o comprometimento de aplicações *Web* legítimas para a inserção de *Web malware* são exemplos de *software* com vulnerabilidades que sofrem abuso com frequência.

As vulnerabilidades em *software* são inseridas acidentalmente, por falta de conhecimento dos programadores ou de controle durante o ciclo de desenvolvimento, durante o processo de codificação, e normalmente envolvem a falta de validação de dados vindos de fontes externas ao sistema. Os métodos mais utilizados para localizar os problemas de segurança das aplicações, para que depois seja possível corrigí-los, são revisões de código

e testes de penetração.

2.2 Técnicas de Ataque

Para realizar ataques contra o navegador *Web* das vítimas, os atacantes precisam fazer com que estas acessem alguma página *Web* que hospede o conteúdo malicioso previamente estabelecido. Isto é alcançado principalmente de duas formas, através de engenharia social e através do comprometimento de páginas legítimas. Para isso, os atacantes muitas vezes fazem uso de *kits* de comprometimento (*exploit kits*), isto é, conjuntos de ferramentas que podem ser utilizados na personalização de ataques e que são encontrados à venda na Internet. As técnicas mencionadas para ataques no lado do cliente são detalhadas a seguir.

2.2.1 Engenharia social

No contexto de segurança da informação, a engenharia social está relacionada com o ato de enganar ou manipular as pessoas para que estas realizem determinadas ações favoráveis ao atacante ou forneçam informações sigilosas. No caso dos ataques a usuários da Internet, a engenharia social acontece principalmente por meio de mensagens de *phishing*. Nestas mensagens, o atacante se passa por outra pessoa ou empresa para tentar levar uma vítima a acessar um determinado *link* ou anexo, os quais geralmente são enviados por *e-mail*, que redirecionará o usuário para uma página contendo o conteúdo malicioso.

A página com conteúdo malicioso pode ter sido hospedada especificamente para essa finalidade ou pode ser o caso de uma página legítima que foi infectada e está hospedando o *Web malware* sem o conhecimento do administrador. Este último caso torna mais difícil a identificação do ataque, dado que o *site* contaminado pode ser considerado "confiável" pela vítima.

2.2.2 Comprometimento de páginas

Devido à grande quantidade de páginas *Web* vulneráveis e a demora na resposta dos administradores para retirar o conteúdo infectado do ar, a hospedagem de conteúdo malicioso através da infecção de páginas *Web* se tornou algo comum. Após inserir tal conteúdo em um *site*, por exemplo, um programa malicioso, o atacante espera que usuários legítimos sejam infectados enquanto o acessam ou envia mensagens de *phishing*, como mencionado na seção anterior.

Em [47] são apresentadas as 10 principais ameaças, atualizadas periodicamente, às aplicações *Web*. A Tabela 2.1 mostra quais são essas ameaças, ordenadas de acordo com

o risco que representam. Destas, as duas primeiras (injeção de código e *cross-site scripting*, ou XSS) podem ser usadas para inserir código malicioso em uma página legítima. Estes dois tipos de ataque são explicados a seguir.

A1	Injection
A2	Cross-Site Scripting (XSS)
A3	Broken Authentication and Session Management
A4	Insecure Direct Object References
A5	Cross-Site Request Forgery (CSRF)
A6	Security Misconfiguration
A7	Insecure Cryptographic Storage
A8	Failure to Restrict URL Access
A9	Insufficient Transport Layer Protection
A10	Unvalidated Redirects and Forwards

Tabela 2.1: As 10 principais ameaças às aplicações *Web*.

Injeção de código

Ataques de injeção de código ocorrem quando a aplicação *Web* falha ao separar o que é um dado do que é código, enquanto processa uma requisição de usuário. Desta forma, um dado aparentemente inofensivo enviado pelo usuário pode ser forçado a ser tratado como código, fazendo com que se consiga mais privilégios sobre o sistema do que era esperado pelo administrador.

O tipo de problema de injeção de código mais comum envolve consultas SQL e o ataque que abusa dele é conhecido como injeção de SQL. Neste caso, um dado enviado pelo usuário é utilizado para formar uma consulta ao banco de dados, fazendo com que este consiga, através da manipulação do referido dado, modificar a consulta e obter informações sobre o banco de dados que não deveriam ser acessíveis. Ainda, é possível que sejam feitas alterações nas informações presentes no banco, dependendo das permissões do usuário configurado na aplicação *Web*.

A Figura 2.1 contém um exemplo de código em linguagem Java que é vulnerável à injeção de SQL. É importante notar que o parâmetro (*nomeProduto*) passado para a aplicação é inserido na consulta sem qualquer tipo de codificação ou validação. Assim, no caso de um usuário passar como parâmetro o valor `' or '1'='1`, a consulta muda para `SELECT * FROM produtos WHERE nomeProd = '' or '1'='1'`, causando a modificação de seu funcionamento original. Vale ressaltar que esse parâmetro em si não causa danos à aplicação, mas outros tipos de requisição poderiam revelar informações que não deveriam ser acessíveis, tais como nomes de usuário e senhas, em certos casos.

```
String consulta="SELECT * FROM produtos WHERE nomeProd='"+request.getParameter("nomeProduto")+"'";
```

Figura 2.1: Código vulnerável a injeção de SQL

Para proteger as aplicações contra problemas de injeção de código é importante efetuar a validação de todo e qualquer dado proveniente de fontes externas. Esta verificação precisa ser feita com o uso de *whitelists*, ou seja, listas de tipos de dados que são permitidos, ao invés de *blacklists*, que são listas do que não é permitido. Pode-se utilizar as duas formas em conjunto, mas é importante não utilizar apenas *blacklists* pois é muito difícil listar todas as possíveis formas de se inserir um dado nocivo que possa comprometer a aplicação. Outra forma de proteção é a tipificação do dado de entrada para que este não seja tratado como código.

Cross-site scripting

Os ataques de XSS são um outro tipo de injeção de código, mas por serem muito frequentes, foram colocados em uma categoria à parte. O *site* xssed [18], que disponibiliza informações sobre este tipo de ataque, possui um arquivo com mais de 39 mil casos de *sites* que são (ou foram) vulneráveis. Nos casos de XSS, como no caso geral de injeção de código, os dados enviados por um usuário são tratados como código, possibilitando a execução de comandos e a consequente efetivação do ataque.

Os ataques de XSS podem ser divididos em três categorias: reflexivos, armazenados e baseados no DOM[1]. No XSS reflexivo, algum dos parâmetros passados para a aplicação *Web* é inserido na resposta enviada ao usuário sem que haja validação ou codificação deste dado. Assim, o atacante faz a vítima acessar um *link* especialmente preparado, o qual aponta para uma página em um servidor vulnerável. Isto pode ser feito através de alguma técnica de engenharia social, conforme descrito anteriormente. Este *link* preparado pelo atacante pode conter um parâmetro com o código malicioso, e, assim que ocorra o acesso por parte da vítima, esta envia para o servidor vulnerável a requisição com o parâmetro malicioso mencionado. Logo, o servidor vulnerável retorna, dentro da resposta enviada, o valor do parâmetro. Desta forma, o código é executado assim que carregado pelo navegador da vítima. Um possível cenário para ilustrar o caso citado é apresentado na Figura 2.2.

Já no ataque de XSS armazenado, o código malicioso é enviado pelo atacante ao servidor-alvo, em meio a dados válidos, fazendo com que este o armazene em um banco de dados, por exemplo. Este código é enviado aos usuários que acessarem esse servidor em uma etapa posterior. No caso de um *blog*, por exemplo, o código poderia ser enviado

[1] *Document Object Model* - estrutura que representa o documento carregado pelo navegador *Web*

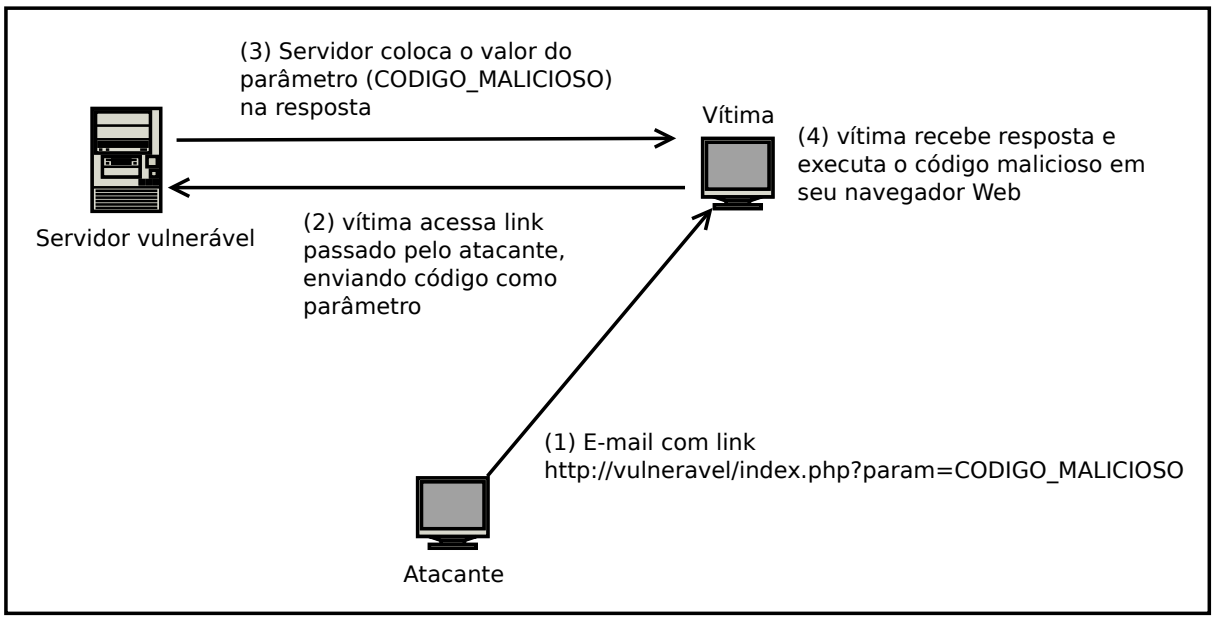

Figura 2.2: Cenário de um ataque XSS reflexivo

pelo atacante junto com um comentário normal. Assim, todos os usuários que porventura visualizem o comentário com código malicioso acabam por recebê-lo e executá-lo. Este exemplo é mostrado na Figura 2.3.

Por fim, há o caso dos ataques baseados no DOM. Nesta situação, o código vulnerável é executado no navegador do usuário e se utiliza de algum dado não verificado (ou validado) para fazer modificações no DOM. Um cenário possível ocorre quando um determinado código que é executado no navegador lê o valor de algum parâmetro da URL e o utiliza para produzir a página vista pelo usuário. Neste cenário, o atacante pode prover um *link* contendo um código malicioso no lugar desse parâmetro, o que resultaria na inserção deste código na página e, consequentemente, em sua execução pelo navegador. O ataque descrito e um exemplo de código vulnerável retornado pelo servidor estão ilustrados na Figura 2.4.

Os ataques de XSS são bastante utilizados no roubo de *cookies*[2], para que o atacante possa ter acesso ao sistema vulnerável com as credenciais da vítima. Devido às restrições de segurança dos navegadores, para ter acesso aos *cookies* de um determinado *site*, o código malicioso precisa ser executado no contexto deste *site*. Por isso, os atacantes precisam utilizar um ataque desse tipo para obter informações dos *cookies*.

As formas de proteção contra ataques do tipo XSS são as mesmas utilizadas para injeção de código, isto é, validação e tipificação dos dados de entrada para verificar que

[2] *Cookies* são conjuntos de dados passados pelas aplicações *Web* aos navegadores e são utilizados para controle de estado em sessões HTTP.

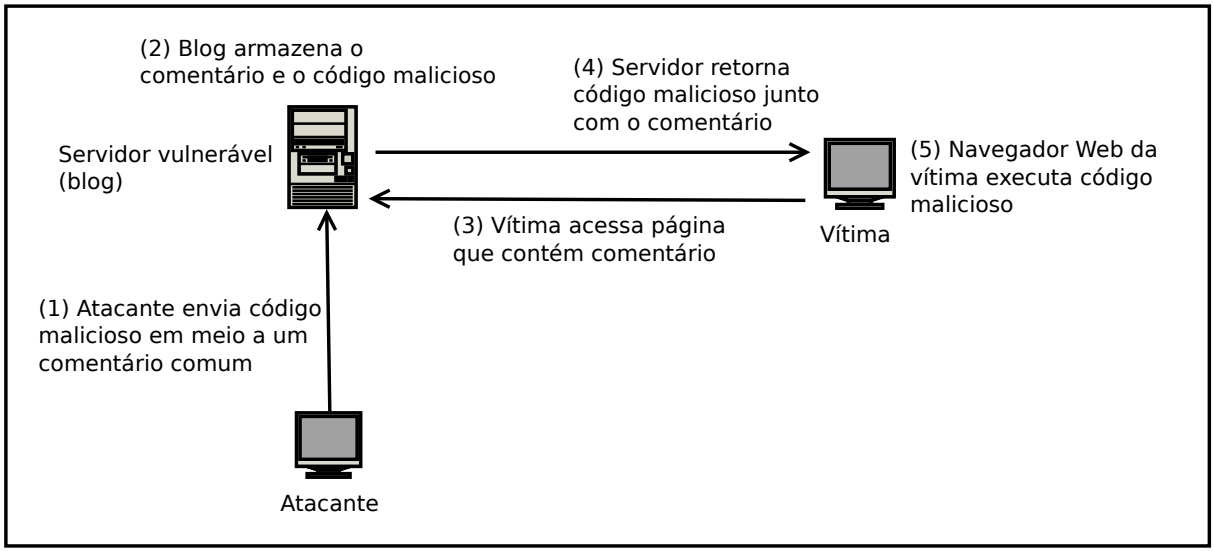

Figura 2.3: Cenário de um ataque XSS armazenado

estes sejam válidos e para que não sejam tratados como código, mas apenas como dados.

2.2.3 Anúncios maliciosos

O uso de anúncios maliciosos, muitas vezes feitos em Flash [20], é uma das formas usadas para levar códigos maliciosos aos navegadores dos usuários. A grande quantidade de anúncios existentes torna a análise manual deles impraticável, implicando no uso de análises automatizadas. Entretanto, as técnicas anti-análise empregadas nos códigos pelos atacantes fazem com que muitos deles passem pelas análises automáticas sem serem detectados. Assim, o administrador de um *site* que hospeda um determinado anúncio pode não saber que os visitantes de seu *site* estão carregando códigos maliciosos e que podem ter seus sistemas comprometidos e seus dados roubados.

2.2.4 Manipulação de mecanismos de busca

Outra forma de levar os usuários a acessar o conteúdo malicioso é hospedá-lo em uma página e fazer com que os mecanismos de busca o coloquem em uma posição alta em sua classificação. Quanto melhor a classificação da página maior a chance de um usuário acabar acessando-a. As principais técnicas usadas para realizar essa manipulação são apresentadas em [26].

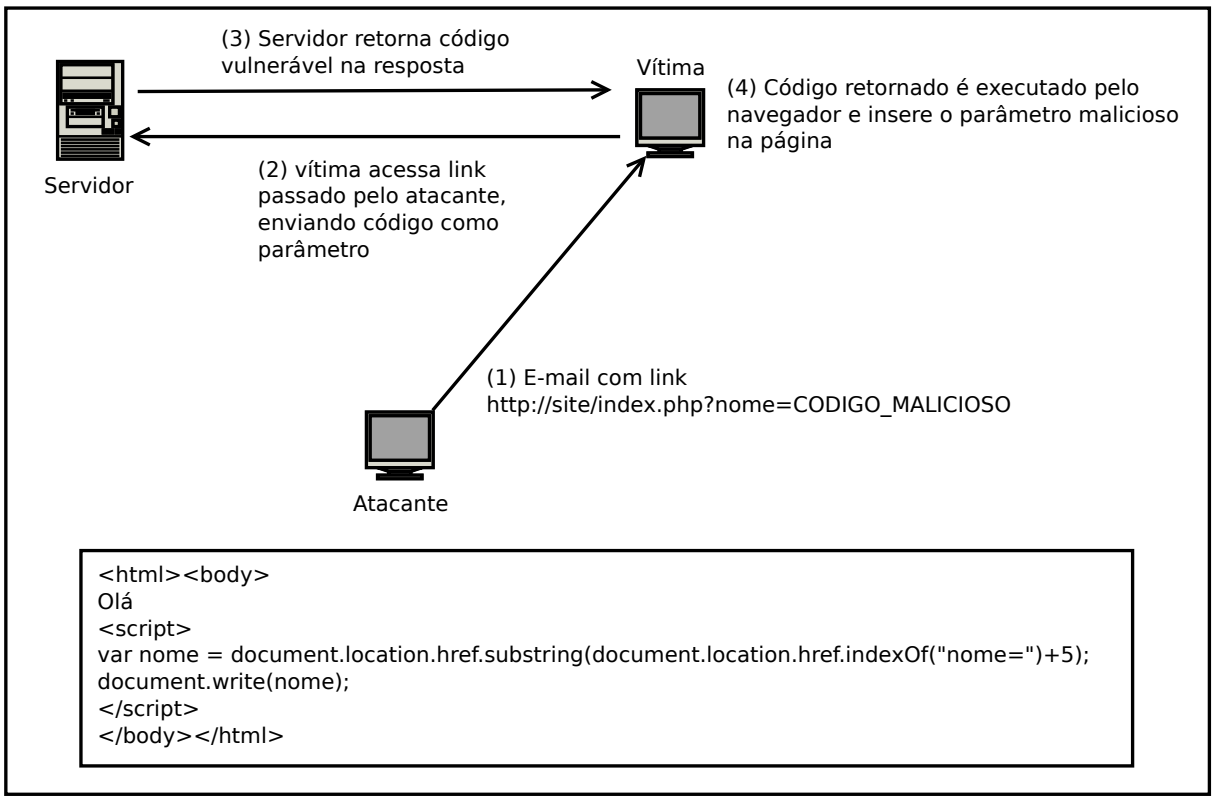

Figura 2.4: Cenário de um ataque XSS baseado no DOM e exemplo de código vulnerável

2.3 *Kits* de comprometimento

Os *kits* de comprometimento são conjuntos de ferramentas que incluem páginas *Web* com conteúdo malicioso, páginas de controle de usuários comprometidas, *malware* para ser instalado no sistema das vítimas e páginas com estatísticas sobre usuários comprometidos. Esses *kits* são vendidos para os atacantes que não querem desenvolver suas próprias ferramentas de abuso ou que não possuem conhecimento suficiente para isso, bastando a eles apenas hospedar essas páginas em algum *site* e levar as vítimas a acessá-lo.

Estes *kits*, em geral, incluem *exploits* para várias vulnerabilidades diferentes, de forma a aumentar a chance de comprometer com sucesso os usuários que acessam a página com conteúdo malicioso. Para citar um exemplo, o *Phoenix exploit kit*, um dos *kits* mais populares atualmente [21], possui, em sua versão 2.3r, *exploits* para as seguintes vulnerabilidades [34]:

- IE MDAC CVE-2006-0003;
- Adobe Flash 9 CVE-2007-0071;

- Adobe Flash 10 CVE-2009-1869;
- Adobe Reader CollectEmailInfo CVE-2007-5659;
- Adobe Reader util.printf CVE-2008-2992;
- Adobe Reader Collab GetIcon CVE-2009-0927;
- Adobe Reader newPlayer CVE-2009-4324;
- Adobe Reader LibTiff CVE-2010-0188;
- Adobe PDF SWF CVE-2010-1297;
- Adobe Reader/Foxit Reader PDF OPEN CVE-2009-0836 ;
- Java HsbParser.getSoundBank (GSB) CVE-2009-3867;
- Java Runtime Environment (JRE) CVE-2008-5353;
- Java SMB CVE-2010-0746;
- IE iepeers CVE-2010-0806;
- Windows Help Center (HCP) CVE-2010-1885;
- IE SnapShot Viewer ActiveX CVE-2008-2463;

Nesta lista de vulnerabilidades, os códigos de CVE são identificadores de vulnerabilidades publicamente conhecidas e catalogadas. Esses códigos podem ser consultados na página *Web* da NVD[3], na qual são mostradas informações a respeito do programa vulnerável, como ele pode sofrer abuso, quais versões são vulneráveis e como corrigir o problema.

Como pode ser observado na lista apresentada, são utilizados *exploits* para diversas aplicações diferentes, aumentando a chance do comprometimento. Além disso, são usados *exploits* tanto para vulnerabilidades antigas (2006) como para vulnerabilidades mais recentes (2010). Isto acontece porque muitos usuários ainda utilizam versões antigas de navegadores e aplicativos, como discutido em [22]. Neste artigo, os autores demonstram que menos de 60% dos usuários do Google utilizam a versão mais atualizada de seus navegadores *Web*.

[3]National Vulnerability Database - http://web.nvd.nist.gov

2.4 Tipos de ataque

Apesar da ameaça mais comum à segurança dos navegadores *Web* ser os ataques do tipo *drive-by download*, que são o foco principal das abordagens existentes para analisar e detectar *Web malware*, existe também a necessidade de detectar códigos cujo objetivo é extrair informações pessoais dos usuários, como histórico de navegação e *cookies*. Ambos os tipos de ataque são explicados a seguir.

2.4.1 *Drive-by downloads*

Ataques do tipo *drive-by download* são normalmente usados para instalar *malware* no sistema da vítima. Estes ataques usualmente precisam de três etapas para serem completados, as quais são detalhadas adiante. A fim de realizar um ataque deste tipo, primeiramente, o atacante precisa carregar um *shellcode* na memória do navegador da vítima. Assim que o navegador acessar este código malicioso, utilizando alguma das abordagens descritas na Seção 2.2, o *shellcode* que está embutido na página é carregado na memória do navegador.

Após isso ocorrer, o código abusa de alguma vulnerabilidade presente no navegador ou em outra aplicação utilizada por ele para completar o ataque, como o interpretador de PDF, biblioteca Java ou Flash player. Em geral, os atacantes empregam uma técnica chamada de *heap-spraying*, explicada mais adiante, para facilitar o processo de abuso. Nestes casos, o tipo de vulnerabilidade mais comum é o *buffer overflow* (estouro de *buffer*), no qual escreve-se mais dados do que há espaço reservado, resultando na sobrescrita de dados de controle, como o endereço de retorno de uma função.

Quando o abuso ocorre com sucesso, o *shellcode* que estava armazenado na memória é executado, o que normalmente resulta no *download* e na execução de um programa malicioso.

Adicionalmente, pode-se utilizar uma outra etapa no início do processo de ataque que consiste na identificação (*fingerprint*) do navegador da vítima, do sistema operacional utilizado e dos componentes instalados. Assim, o código malicioso verifica se possui *exploits* para alguma das aplicações identificadas e tenta realizar o abuso apenas destas aplicações. Isto evita a execução de ataques desnecessários que podem facilitar a detecção e análise do código de ataque ou resultar no mau funcionamento do navegador, impedindo que algum dos outros *exploits* que poderia comprometer o sistema com sucesso seja executado.

Quase todos os navegadores *Web* têm a possibilidade de utilizar componentes extras (*plug-ins*), os quais são desenvolvidos por outras empresas ou usuários para estender suas funcionalidades. Estes componentes possuem os mesmos privilégios do navegador, podendo modificar arquivos e iniciar processos, por exemplo, e normalmente são menos testados do que o navegador *Web* em si, portanto mais propensos a possuir vulnerabilidades.

Em certos casos, como no caso do Internet Explorer, estes componentes são desenvolvidos em linguagens inseguras[4], por exemplo C, e portanto são mais propensos a possuir vulnerabilidades típicas deste tipo linguagem, como o buffer overflow. Além disso, estes componentes compartilham o espaço de memória do navegador, facilitando a execução de ataques como o *heap-spraying*, em que o *shellcode* está localizado na memória do navegador. Pela soma de todos esses fatores, os navegadores se tornaram um alvo interessante para os ataques de *drive-by-download* [15].

Outra forma utilizada por atacantes para instalar *malware* no sistema das vítimas é através de APIs[5] desenvolvidas de forma insegura. Certas APIs, para facilitar o funcionamento do componente a que pertencem ou para disponibilizar mais funcionalidades aos usuários, criam funções que, se não forem protegidas de forma correta, podem ser usadas de maneira maliciosa por atacantes.

Um exemplo deste problema é o caso do método *DownloadAndInstall* do componente ActiveX Sina [19]. Este método possui um parâmetro que é uma URL que indica um arquivo para ser obtido e instalado no sistema. Originalmente, esse componente foi desenvolvido para facilitar a instalação de atualizações do próprio, mas acabou sendo utilizado em muitos códigos maliciosos para instalar *malware* no sistema da vítima.

Heap-spraying

Diversas técnicas foram criadas para combater ataques de buffer overflow, incluindo técnicas aplicadas no compilador, como StackGuard [10] (usa canários para verificar se a pilha foi sobrescrita), e técnicas aplicadas no sistema operacional, como ASLR [44] (aleatorização da área de memória de partes do arquivo executável) e DEP [33] (impede que certas áreas de memória sejam usadas para execução de código e protege contra a sobrescrita de certas estruturas do Windows). Devido a estas técnicas, os ataques mais tradicionais (baseados na *stack* e no *heap*) ainda existem, mas a aplicação deles se tornou mais difícil, dando espaço para o uso em massa de outro método de abuso chamado de *heap-spraying*.

Com o *heap-spraying*, o atacante insere o *shellcode* muitas vezes na memória, visando aumentar a chance de um ataque que causa um redirecionamento do fluxo de execução do navegador para alguma localização do *heap* que execute este código. Como no navegador os adversários podem utilizar *scripts* (em linguagem JavaScript, por exemplo) para alocar objetos na memória com facilidade, este tipo de ataque se tornou comum para abusar

[4] A linguagem em si não é realmente insegura, mas ela deixa verificações de segurança, como a verificação se uma certa quantidade de dados irá além do limite do buffer escrito, a cargo do programador. Por outro lado, existem linguagens, como Java, que fazem essa verificação sem que o programador precise se preocupar com ela

[5] *Application Programming Interface*

```
01. var payload = unescape("... SHELLCODE CODIFICADO ...");
02. var nop = unescape("%u9090%u9090%u9090");
03. var bigblock = nop;
04. while(bigblock.length < 0x40000){
05.     bigblock += bigblock;
06. }
07. var mem_array = new Array();
08. for(var i = 0; i < 1400; i++){
09.     mem_array[i] = bigblock + payload;
10. }
```

Figura 2.5: Código que realiza *heap-spray*

deles.

Para aumentar a chance do *shellcode* ser executado corretamente após o redirecionamento, os atacantes inserem um grande trecho consecutivo de código, conhecido como *NOP sled*, antes do *shellcode*. Este trecho é composto por instruções simples de máquina, como instruções NOP[6] e instruções de incremento e decremento de registradores, por exemplo. A execução deste tipo de trecho não altera significativamente o estado do sistema, levando à execução do *shellcode* independentemente da parte do *NOP sled* no qual a execução é iniciada. A Figura 2.5 apresenta um código em linguagem JavaScript que implementa o *heap-spraying*.

No código da figura, as seguintes etapas são processadas:

- Na linha 1 o *shellcode* é decodificado e colocado em uma variável.
- Entre as linhas 2 e 6 é criado o *NOP sled*, constituído de instruções NOP (*byte* 0x90).
- Nas linhas 7 até 10, são alocados na memória 1400 objetos, cada um contendo um trecho de *NOP sled* seguido do *shellcode*.

2.4.2 Roubo de informações pessoais

O roubo de informações pessoais é outro tipo grave de ataque, dado que elimina a confidencialidade dos dados dos usuários e é uma das principais motivações para crimes cibernéticos. Neste trabalho, o foco foi direcionado para *cookies* e histórico de navegação, mas esse tipo de ataque poderia ser aplicado contra qualquer tipo de informação pessoal armazenada pelo navegador *Web*, tais como informações colocadas em formulários.

[6]No Operation, isto é, instruções que não realizam nenhuma operação.

O roubo do histórico de navegação viola a privacidade dos usuários, enquanto os *cookies* podem ser usados para que o atacante se passe pelo usuário ao acessar determinadas páginas *Web*. Os códigos utilizados para executar ataques de *drive-by download* e roubo de histórico de navegação podem ser executados a partir de qualquer página infectada. Entretanto, o código que realiza roubo de *cookies* precisa ser executado no contexto do *site* cujo *cookie* deseja-se extrair, devido à segurança interna dos navegadores. Sendo assim, esse tipo de código precisa estar envolvido com ataques de *cross-site scripting*, explicados anteriormente na Seção 2.2.

2.5 Técnicas anti-análise

Para dificultar a análise de seus códigos maliciosos por parte de pesquisadores de segurança e empresas que desenvolvem mecanismos de proteção, os atacantes empregam diversas técnicas no próprio código ou no servidor que o hospeda. A seguir, serão descritas algumas destas técnicas, divididas entre as que são embutidas no código malicioso e as que se encontram no servidor hospedeiro.

2.5.1 Técnicas embutidas no código

Este tipo de técnica consiste em modificações nos códigos, tornando-os mais difíceis de se analisar tanto de forma manual quanto de forma automática. Para impedir a análise manual são feitas muitas modificações no código, tornando esta análise impraticável sem o uso de ferramentas automatizadas. Por outro lado, as técnicas que impedem a análise automática se aproveitam de diferenças inerentes aos ambientes reais e automatizados, como será visto a seguir.

Ofuscação

A ofuscação é utilizada para transformar o código, mantendo a funcionalidade original, e deixar sua leitura e análise mais complexa. Esta técnica inclui muitas operações de manipulação de *strings* e execução de código dinamicamente. A execução dinâmica de código consiste da execução do conteúdo de uma variável que contém texto. Além disso, algumas páginas incluem técnicas de criptografia utilizando a URL como chave para decifrar. Assim, caso o código esteja sendo executado a partir de outra URL, diferente da original onde o atacante hospedou o conteúdo malicioso, sua execução irá terminar com erro e o comportamento malicioso não será executado, dificultando o estudo do código.

Interação com o DOM

Diversos *Web malware* necessitam de interação com o DOM para executar a parte maliciosa do código. Com isso eles dificultam a análise feita em sistemas de análise que se utilizam de emuladores de navegador *Web* e que não possuem a emulação do ambiente completo do navegador. A manipulação da propriedade *innerHTML*, por exemplo, pode ser usada para evadir a análise de dois dos sistemas de análise de *Web malware* mais conhecidos, o PhoneyC [38] e o JSand [9]. O código JavaScript apresentado na Figura 2.6 pode ser usado para efetuar tal evasão.

```
1. var codigo_malicioso = "<script>...</script>";
2. document.getElementsByTagName("body")[0].innerHTML += codigo_malicioso;
```

Figura 2.6: Código JavaScript que realiza manipulação da propriedade *innerHTML* para evasão de sistemas de análise

O código de evasão apresentado efetua as seguintes operações: na linha 1, o código malicioso, colocado entre *tags script*, é alocado em uma variável, enquanto que na linha 2 este código é inserido na página através da modificação da propriedade *innerHTML* do elemento *body*.

Interação com usuário

Alguns códigos maliciosos podem esperar por interações do usuário para ativar o comportamento malicioso. Para isso eles fazem uso de eventos que precisam da interação do usuário para serem gerados, por exemplo *onMouseOver*, que resulta em uma atividade quando o ponteiro do *mouse* encontra-se sobre determinado objeto. Como os sistemas de análise são executados automaticamente, estes eventos não são gerados e o comportamento malicioso não é executado.

Tempo de espera

Outra técnica aplicada nos códigos maliciosos é a espera de um determinado tempo antes de executar o comportamento malicioso. Não são utilizados valores grandes de tempo, pois assim o usuário poderia deixar a página antes que o código malicioso fosse executado. Entretanto, caso o sistema de análise seja executado em um período de tempo menor do que o tempo de espera do código, não será capaz de detectar o comportamento malicioso.

Objetos ActiveX falsos

Certos sistemas de análise de *Web malware*, como o JSand [9], emulam cada objeto ActiveX requisitado pelo código analisado. Este tipo de abordagem é útil para que o sistema seja capaz de analisar códigos que atacam os mais diversos componentes ActiveX existentes, mas torna o sistema facilmente detectável e contornável. O código apresentado na Figura 2.7 é capaz de detectar e evadir a análise de um sistema que emula todo objeto ActiveX requisitado, através da criação de um objeto que não existiria em um navegador *Web* completo. Em um ambiente real, a tentativa de criar um objeto inexistente gera uma exceção e o fluxo de execução do código é desviado para a cláusula *catch*, levando à execução do código malicioso. Entretanto, como o sistema de análise emula todo objeto ActiveX requisitado, essa ação não iria gerar uma exceção e a execução iria continuar normalmente, causando a execução de um código inofensivo.

```
1. try{
2.              var obj = new ActiveXObject("ObjetoQueNaoExiste");
3.              /* está dentro do sistema de análise, executa código benigno */
4. }catch(e){
5.              /* não está no sistema de análise, executa código malicioso */
6. }
```

Figura 2.7: Código JavaScript usado para detectar e evadir sistemas que fazem emulação de todos os objetos ActiveX requisitados pelo código

2.5.2 Técnicas aplicadas no servidor

Estas técnicas são empregadas no servidor que está hospedando o código malicioso, tendo por objetivo diferenciar as vítimas, de forma a identificar usuários que estão tentando acessar o código para análise (detecção) ou focar o ataque a determinado grupo, por exemplo, de um determinado país.

Bloqueio por localização

Uma das abordagens no lado do servidor utilizada atualmente para dificultar a análise dos códigos maliciosos é a verificação do país de origem dos usuários acessando o site. Nestes casos, o conteúdo malicioso é retornado apenas para requisições vindas de determinados países [5]. Assim, se o sistema que está analisando a página não for do país alvo do ataque, não será capaz de analisá-lo.

Limitação de acesso

Em certos casos, a aplicação *Web* que está distribuindo o código malicioso retorna tal conteúdo apenas uma vez para cada endereço IP, retornando páginas sem o conteúdo malicioso em requisições subsequentes [14]. A limitação de acesso ocorre devido ao pressuposto que, se uma vítima já acessou o conteúdo malicioso, ou está comprometida ou não possui as vulnerabilidades que poderiam sofrer abuso. Em ambos os casos, um novo envio do conteúdo malicioso não funciona. Entretanto, o mais importante é que tal limitação dificulta o trabalho de pesquisadores que desejam estudar o código malicioso nestes *sites*, pois cada endereço IP utilizado pode acessá-lo apenas uma vez.

Bloqueio de endereços IP

AV Tracker[7] é uma lista de endereços IP de empresas de anti-vírus, grupos de pesquisa em segurança e sistemas de análise de *malware* conhecidos publicamente. Esta lista serve aos atacantes tanto para possibilitar a codificação de mecanismos anti-análise como para dificultar o acesso destes grupos de segurança ao código malicioso. Com isso, o código que se utiliza deste tipo de lista de bloqueio pode não ser analisado em tempo hábil, atrasando sua detecção e criação de contra-medidas por parte das ferramentas de proteção existentes.

Verificação do *referrer*

O *referrer* é um dos cabeçalhos do protocolo HTTP que indica o *site* anterior que o navegador estava visitando antes de efetuar a requisição atual.

Quando o atacante tem controle sobre os *sites* nos quais suas URLs maliciosas estão sendo anunciadas, ele pode fazer a verificação da informação de *referrer* nas requisições enviadas para seu servidor. Se estas requisições contêm um valor diferente do esperado, o servidor responde com um conteúdo inofensivo, retornando o conteúdo malicioso apenas quando a informação estiver correta. Dessa forma, o atacante diminui a chance de que um indivíduo que obtenha a URL que contém o código malicioso em listas compartilhadas pela comunidade de segurança consiga o código para efetuar análises.

2.6 Conclusão

Neste capítulo foram apresentadas as principais metodologias empregadas pelos atacantes para fazer com que os usuários acessem o conteúdo malicioso, que são a engenharia social, o comprometimento de páginas legítimas, o uso de anúncios maliciosos e a manipulação de mecanismos de busca. Em seguida foram detalhados os ataques (roubo de informações

[7] http://honeyblog.org/archives/37-AV-Tracker.html

e *drive-by download*) usados contra os navegadores *Web*, e por fim foram mostradas as técnicas usadas pelos atacantes para dificultar a análise de seus códigos por parte de grupos de pesquisa e empresas que desenvolvem mecanismos de segurança, incluindo códigos que são capazes de evadir a análise de certos sistemas de análise de *Web malware* atuais.

Capítulo 3

Análise e proteção

Uma parte relevante da pesquisa aplicada à segurança *Web* está relacionada à análise e detecção de *Web malware*. Estes trabalhos diferem principalmente em relação ao tipo de ataque tratado, ao tipo de código analisado e se podem ser utilizados para proteção ou se são apenas informativos.

Neste capítulo são descritos os principais sistemas de análise de *Web malware* existentes, bem como os mecanismos de proteção contra os ataques descritos no capítulo anterior. São apresentadas as abordagens que atuam no lado do cliente, no lado do servidor e que atuam em ambos.

3.1 Sistemas de análise de *Web malware*

Existem diversas abordagens para a análise de *Web malware*, que diferem quanto ao tipo de ataque analisado e de código suportado. Parte dos sistemas é capaz de analisar apenas ataques que utilizam JavaScript, enquanto outros detectam apenas os que completam o processo de abuso com sucesso.

Os sistemas que analisam *Web malware* são conhecidos como *honeyclients* e são divididos em dois grupos, de acordo com a forma como processam a página analisada. Aqueles que utilizam um emulador para carregar o conteúdo *Web* são chamados de *honeyclients* de baixa interatividade. Já os que utilizam um navegador *Web* completo, dentro de algum sistema emulado ou virtualizado, são chamados de *honeyclients* de alta interatividade.

Estes sistemas são utilizados em modo *offline* para produzir informações a respeito dos códigos maliciosos e não podem ser utilizados em produção para prover proteção aos sistemas. As informações providas são de grande importância no estudo de ataques para que sejam analisadas as técnicas empregadas nos códigos maliciosos e seus comportamentos. Com estas informações, as empresas que desenvolvem programas de proteção podem assim atualizar seus sistemas de acordo, provendo uma segurança maior aos usuários.

3.1.1 Análise utilizando emuladores

Os *honeyclients* de baixa interatividade utilizam emuladores de navegadores para tratar as páginas *Web* analisadas. Como esses sistemas não executam de fato o código analisado, não há o comprometimento do sistema, incorrendo na não necessidade de restauração após o processo de análise. Por isso, este tipo de *honeyclient* costuma ser mais rápido do que os de alta interatividade. Entretanto, por não utilizarem navegadores *Web* reais, com o ambiente completo (características, *plug-ins*) que os navegadores possuem, estes sistemas são mais facilmente detectados e evadidos. Isto pode ser feito através da utilização de técnicas descritas na Seção 2.5.1. Atualmente, os principais sistemas de análise de baixa interatividade são JSand [9] e PhoneyC [38].

JSand

JSand é um sistema disponível publicamente para uso através de sua página de submissão[1]. Sua função principal é analisar códigos escritos na linguagem JavaScript, apresentar informações a respeito do comportamento obtido e indicar se o código em questão é malicioso, suspeito ou benigno. A identificação de comportamentos maliciosos é feita por meio da detecção de anomalias. Para emular o ambiente do navegador, JSand utiliza o HtmlUnit[2], uma plataforma em Java para teste de aplicações *Web*. Para aumentar a abrangência de códigos analisados ele emula a existência de cada objeto ActiveX requisitado pelo código. Dada a existência de códigos que abusam de diversos componentes diferentes, a emulação de objetos ActiveX permite que o sistema analise uma quantidade maior de códigos, mas, como citado na seção 2.5.1, torna-o de fácil detecção e evasão.

A detecção por anomalia é feita com o uso de dez atributos extraídos do código JavaScript que apresentam características de redirecionamento, ofuscação, desofuscação, preparação do ambiente para o abuso e processo de abuso. Para realizar o treinamento e detecção JSand utiliza a biblioteca libAnomaly[3], desenvolvida pelo mesmo grupo que desenvolveu o JSand.

Este sistema provê diversas informações a respeito do código analisado, além da classificação como benigno, suspeito ou malicioso. Estas informações incluem trechos de código desofuscado, vulnerabilidades que sofreram abuso, *shellcodes* utilizados, objetos ActiveX usados, *links* para o sistema Anubis[4] (com a análise de arquivos executáveis que podem ser instalados após o comprometimento) e requisições HTTP efetuadas.

Além do problema da detecção e evasão do ambiente de análise, o JSand é limitado à análise apenas de ataques de *drive-by download* e que utilizam a linguagem JavaScript.

[1] http://wepawet.cs.ucsb.edu/
[2] http://htmlunit.sourceforge.net/
[3] http://www.cs.ucsb.edu/~seclab/projects/libanomaly/
[4] http://anubis.iseclab.org/

Porém, mesmo com os problemas apresentados, é o sistema de análise disponível publicamente mais conhecido, utilizado e referenciado na área.

PhoneyC

Outro sistema que utiliza um emulador para processar as páginas a serem analisadas é o PhoneyC [38]. Ele é capaz de analisar códigos JavaScript e Visual Basic Script (VBS). A emulação do ambiente de JavaScript é feita com o uso do interpretador da Mozilla—SpiderMonkey[5]—enquanto que a do VBS é feita com a ferramenta vb2py[6], que transforma código VBS em equivalente em Python, o qual é então analisado com um interpretador próprio da linguagem.

PhoneyC emula a existência de determinados objetos ActiveX e é capaz de detectar ataques que tentam abusar deles. As informações providas sobre o código analisado estão relacionadas com objetos ActiveX usados, vulnerabilidades que sofreram abuso e *shellcodes* utilizados.

Por não emular todos os objetos ActiveX que podem ser requisitados, PhoneyC não está vulnerável ao ataque que instancia um objeto inexistente, apresentado na Seção 2.5.1. Entretanto, por não possuir um ambiente de navegador completo, sua análise pode ser evadida com a técnica que utiliza a propriedade *innerHTML*, esta apresentada na Seção 2.5.1. Além disso, PhoneyC é capaz de detectar apenas ataques de *drive-by download* e que utilizam JavaScript ou VBS, deixando de analisar os que fazem uso de Java, por exemplo.

3.1.2 Análise utilizando navegador *Web* real

Os sistemas de análise que se utilizam da tecnologia de *honeyclients* de alta interatividade processam as páginas, em geral, dentro de um ambiente virtualizado ou emulado (entenda-se aqui o conceito de ambiente como sendo o do sistema operacional). Dentro deste tipo de sistema é utilizado um navegador completo e são coletadas informações a respeito do código analisado ou do comportamento do navegador. As informações coletadas são então processadas para descobrir se houve comportamento malicioso. A seguir, são apresentados sistemas que se baseiam em *honeyclients* de alta interação e cuja implementação utiliza um *driver* de *kernel* ou uma técnica conhecida como *hooking* de API para capturar as informações necessárias para a análise.

[5]SpiderMonkey é um interpretador de JavaScript mantido pela Mozilla Foundation que faz parte do navegador *Web* Firefox. Mais informações sobre ele podem ser encontradas em http://www.mozilla.org/js/spidermonkey/.

[6]http://vb2py.sourceforge.net

Capture-HPC

Capture-HPC [43] é um *honeyclient* preparado para visitar e monitorar páginas *Web* através da análise de chamadas de sistema. Dentro de uma máquina virtual, ele utiliza um navegador *Web* completo para acessar as páginas a serem analisadas e um *driver* de *kernel* de Windows para monitorar as chamadas de sistema feitas pelo navegador e por processos iniciados por ele. Essa monitoração é realizada com o uso de *kernel callbacks*, que fazem com que o *driver* seja informado quando certas ações são realizadas pelo *kernel*. As chamadas de sistema capturadas são analisadas posteriormente e, caso alguma seja considerada anômala, a página sob análise é marcada como maliciosa. As chamadas de sistema anômalas são todas as que não estiverem em uma lista previamente estabelecida de chamadas autorizadas.

A detecção de comportamento malicioso através da monitoração de chamadas de sistema permite a detecção de ataques de *drive-by download* independentemente da linguagem de *script* utilizada para realizar a exploração. Entretanto, ela só permite a detecção dos ataques que forem bem sucedidos, ou seja, que resultem na execução de chamadas de sistema anômalas.

Web Exploit Finder

WEF (Web Exploit Finder) [36] é um sistema semelhante ao Capture-HPC, por utilizar a tecnologia de máquina virtual com um *driver* de *kernel* de Windows para monitorar chamadas de sistema. A diferença está na técnica usada para tal. Enquanto o Capture-HPC monitora as chamadas de sistema com o uso de *kernel callbacks*, o WEF utiliza *hooking* de SSDT. Esta abordagem modifica a tabela que contém o endereço de certas funções nativas do sistema operacional, de forma que as funções do *driver* que registram as ações sejam chamadas antes.

Shelia

O sistema Shelia [42] processa *e-mails* para encontrar *links* e arquivos enviados em anexos. Então, com base no tipo de arquivo (texto, PDF, arquivo do Microsoft Office etc.) ou se um *link* for encontrado, Shelia passa o objeto para uma aplicação específica que o processa. Os *links* são encaminhados para um navegador *Web*. Assim que a aplicação é iniciada, uma biblioteca do Windows (DLL - *Dynamic Link Layer*) é injetada em sua memória. Essa DLL é responsável por fazer *hooking* de certas APIs, que incluem funções de modificação de arquivos, modificações de registro, criação de processos, entre outras. Estes *hooks* tratam-se de modificações que fazem com que as funções da DLL sejam chamadas antes das funções nativas, registrando as operações efetuadas. Para detectar os comportamentos maliciosos, o sistema verifica a área de memória que originou as ações e, quando estas são

originárias de uma área que não deveria ser executada, um ataque é detectado. Quando isto acontece, o sistema bloqueia a execução do comportamento malicioso. O problema da abordagem é não poder detectar ataques de roubo de informações, além de poder ser evadida com os métodos apresentados em [6].

3.2 Sistemas de proteção

Outros sistemas realizam a análise do código malicioso em tempo real e podem ser usados para bloquear ataques antes que estes comprometam o sistema dos usuários, ao invés de apenas prover informações sobre eles e classificá-los.

Estes sistemas de proteção podem ser divididos entre os que atuam isoladamente no lado cliente e os que atuam no lado cliente em conjunto com o lado servidor. Ambos os tipos são apresentados nesta seção. Como a injeção de código malicioso em páginas legítimas é uma das principais formas usadas para que o usuário carregue o *Web malware*, também são apresentados sistemas que protegem os servidores de aplicação *Web*.

3.2.1 Proteções no lado cliente

Diversos sistemas podem ser usados para proteger os usuários enquanto navegam na *Web*. Estes são formados por extensões de *kernel*, sistemas que atuam em *proxies*, sistemas integrados diretamente ao navegador, listas de bloqueio e novas arquiteturas de navegadores *Web*, os quais são descritos a seguir.

Extensão de *kernel*

BLADE [32] é uma extensão de kernel usada para bloquear ataques do tipo *drive-by download*. Seu objetivo é impedir que arquivos binários executáveis obtidos pelo navegador sem a autorização do usuário sejam executados no sistema operacional da vítima. Entretanto, essa abordagem possui os seguintes problemas: não é capaz de detectar ataques que não sejam do tipo *drive-by download* e de impedir a execução de *scripts* ou códigos que sejam executados diretamente da memória sem que o executável seja escrito em disco.

Proxies

Outras abordagens de proteção são agregadas a *proxies*, podendo modificar o código das páginas acessadas ou analisá-lo e bloqueá-lo, caso seja detectado como malicioso, antes que chegue ao navegador do usuário.

Em [41] os autores apresentam Cujo, um sistema que extrai atributos estáticos e dinâmicos do código JavaScript, emulando-o através do SpiderMonkey. A detecção do

comportamento malicioso é feita com o uso de um classificador que utiliza *support vector machines* (SVM), uma técnica bem específica de inteligência artificial que se baseia em redes neurais. Quando um código é identificado como malicioso, é bloqueado e não alcança o usuário. As limitações desta abordagem estão relacionadas a tratar apenas JavaScript e ataques de *drive-by download*.

Webshield [31], por sua vez, utiliza um emulador dentro do *proxy* e um agente no sistema do usuário. O emulador executa o código JavaScript e verifica comportamentos maliciosos, bloqueando-os quando a detecção ocorre. Se o comportamento malicioso não é detectado, o conteúdo do navegador do usuário é atualizado. Webshield é capaz de detectar tanto ataques de *drive-by download* quanto ataques que roubam informações do usuário. Entretanto, não pode detectar ataques que utilizam outras linguagens, como Flash, Java e VBScript.

Em [35], os autores apresentam Spyproxy, um sistema que consiste de uma máquina virtual com um navegador *Web* e mecanismos que detectam o abuso deste. As páginas acessadas pelo usuário passam primeiro pela máquina virtual e, caso não seja detectado comportamento malicioso, são encaminhadas ao usuário. Porém, tal abordagem não detecta ataques que roubam informações dos usuários.

BrowserShield [40] é outro sistema que pode ser implementado em um *proxy*. Seu funcionamento implica em modificar o código JavaScript enviado ao cliente de forma a fazê-lo executar funções confiáveis. Isto é feito através de políticas para mediar o acesso de código considerado não-confiável à estrutura da página carregada. Um exemplo de política é o uso de assinaturas para detectar a utilização de *exploits* conhecidos. Entretanto, este sistema detecta apenas problemas gerados por JavaScript.

Sistemas integrados no navegador *Web*

Existem também sistemas que são integrados ao navegador *Web* que, para capturar as ações do código JavaScript, interceptam o processador da linguagem nativo do navegador ou utilizam outro interpretador de JavaScript instrumentalizado.

Em [15], os autores apresentam um sistema que realiza modificações no processador de JavaScript SpiderMonkey de forma a monitorar a criação de objetos do tipo *string* e verificar, utilizando a libemu [2], a existência de *shellcodes* na memória. O problema deste sistema, similarmente à grande parte dos mencionados anteriormente, é ser capaz de analisar apenas ataques que utilizam JavaScript e do tipo *drive-by download*.

Zozzle [12] é um sistema que intercepta as chamadas para a função *Compile* da biblioteca do Internet Explorer (*jscript.dll*) que processa JavaScript, analisando todo o código interpretado pelo navegador. Esta análise é predominantemente estática e nela são extraídos atributos para classificar o código como malicioso ou benigno, com o uso de um classificador *naïve Bayes*. Este sistema também é capaz de detectar apenas ataques que

utilizam JavaScript e do tipo *drive-by download*.

ADSandbox [13] analisa as páginas *Web* antes que sejam passadas para o navegador. Com o uso do SpiderMonkey, ADSandbox registra as ações executadas pelo código JavaScript e procura por padrões de comportamento malicioso. Com isso, é capaz de detectar tanto ataques de *drive-by download* quanto ataques que roubam informações do usuário. Entretanto, ADSandbox analisa apenas códigos escritos em linguagem JavaScript e não é capaz de analisar códigos que usam modificações no DOM para ofuscação.

Além do ADSandbox, outro sistema integrado ao navegador que pode detectar ataques de roubo de informações do sistema é apresentado em [46]. Este sistema utiliza *data tainting* para traçar o fluxo das informações a serem protegidas dentro do navegador e bloqueia ataques de extração de informação antes que estas informações sejam enviadas para outro domínio, solicitando ao usuário a autorização do procedimento de envio.

Listas de bloqueio

Outra abordagem comumente utilizada por navegadores *Web* para proteger os usuários contra páginas maliciosas são as *blocklists*, listas que contém endereços de sites onde foi encontrado conteúdo malicioso. O *SmartScreen Filter*[7] é uma funcionalidade implementada no navegador Internet Explorer que, enquanto o usuário navega na Internet, verifica se as páginas acessadas fazem parte de alguma das *blocklists* configuradas, bloqueando o acesso nos casos positivos. Já a *Safe Browsing API*[8] do Google trata-se de um serviço usado por outros navegadores *Web* para informar se uma dada URL contém código malicioso. O serviço verifica a URL em sua lista, a qual é constantemente atualizada, e informa o usuário de acordo com o resultado da consulta. Para manter essas listas atualizadas, as empresas responsáveis por elas precisam estar constantemente verificando as páginas da Internet com seus sistemas de análise.

Novas arquiteturas de navegador

Os ataques existentes ocorrem devido aos problemas encontrados na arquitetura ou na implementação dos navegadores e componentes *Web*. Por isso, foram desenvolvidos estudos para criar novas arquiteturas de navegador *Web*, mais seguras do que as atuais. Em [23], é proposta uma arquitetura que utiliza mecanismos no nível do sistema operacional para isolar o navegador e cada uma das páginas acessadas por ele. Já em [11], este isolamento é obtido através do uso de virtualização. Tais projetos, apesar de aumentarem o nível de segurança do usuário utilizando novas arquiteturas, também podem conter problemas de

[7] http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter
[8] http://code.google.com/apis/safebrowsing/

implementação decorrentes da não consideração adequada dos requisitos de segurança no ciclo de desenvolvimento.

3.2.2 Proteções no cliente e no servidor

Uma das principais dificuldades do uso de mecanismos de proteção no lado cliente que bloqueiam a execução de código malicioso é saber em quais partes das páginas tais códigos podem estar inseridos, bem como qual tipo de atividade é executado por cada aplicação. Em uma aplicação na qual o código JavaScript não precisa acessar *cookies*, o acesso pode ser completamente bloqueado no navegador, por exemplo. Este tipo de informação pode, em geral, ser disponibilizado pelos desenvolvedores das aplicações. Para isso foram criadas técnicas em que o servidor passa para o navegador informações referentes a políticas de execução de código, para que estas sejam aplicadas e haja o impedimento da execução de código malicioso. De modo a forçar o código executado no navegador a seguir essas políticas, são feitas modificações no código JavaScript antes que este chegue ao navegador, ou são feitas modificações no código-fonte do navegador.

Modificações no código fonte do navegador

Em [37, 25] os autores diferenciam conteúdo confiável e não-confiável no servidor e o navegador utiliza essa informação para sua proteção, tentando preservar a integridade da estrutura do documento que está carregado no navegador. Em [37], o foco se dá na proteção de forma dinâmica, enquanto Noncespaces [25] tem seu foco no uso de políticas mais detalhadas.

Outra abordagem existente é o uso de funções que aplicam no navegador as políticas definidas pelo servidor. No método apresentado em [29], o servidor envia uma função em conjunto com políticas ao navegador, de forma que este sempre execute a função enviada antes de cada chamada feita a um código em JavaScript. Os autores propõem o uso de políticas de *whitelisting*, em que apenas códigos cujo *hash* está na lista são executados. Entretanto, a geração desta lista se torna difícil nos casos em que o código é gerado automaticamente, ou que não se tenha conhecimento dele previamente, ou ainda, se houver muitos blocos pequenos de código. Além disso, atacantes podem usar métodos que estão na lista para realizar ataques, como no caso do *XMLHttpRequest* (muito usado em aplicações *Web* 2.0) que pode ser usado para enviar o *cookie* do usuário para outro site. Em [16], Erlingsson et al. propõem uma abordagem em que funções JavaScript são executadas a cada transformação do documento, ao invés de antes da execução de blocos de código. Esta abordagem possui uma flexibilidade maior, ou seja, pode usar políticas mais abrangentes e detectar atividades maliciosas mais variadas do que a anterior.

Modificações no código JavaScript

Também existem estudos cujo foco é em métodos para alterar o código JavaScript que é enviado ao navegador, como em [39]. Neste, os autores propõem a alteração do código enviado ao cliente de forma que seja possível aplicar políticas e não seja necessário fazer modificações no código-fonte do navegador. Esta técnica pode ser usada com políticas definidas no servidor, protegendo-o de possíveis códigos injetados indevidamente na aplicação. Entretanto, neste caso, as políticas não são aplicadas a códigos de outros domínios carregados através de *frames* ou *iframes*. Uma outra utilização pode ser sob a forma de um *proxy* ou um componente no navegador, com políticas definidas pelo usuário. Esta ferramenta também é capaz de detectar apenas ataques que utilizam código JavaScript.

O fator que torna os mecanismos propostos mais efetivos também é uma desvantagem deles, independentemente de utilizarem modificações no código ou no navegador *Web*. O servidor pode criar políticas que definem o comportamento válido do código que é executado no cliente de forma mais precisa, porém, isso significa que o usuário fica dependente da criação dessas. Caso as políticas não sejam corretamente definidas ou o servidor for comprometido e tiver suas políticas modificadas, o problema permanece.

3.2.3 Proteção no servidor

Como dito anteriormente, a injeção de código malicioso em aplicações legítimas é um dos principais meios utilizados por atacantes para fazer com que os usuários acessem o *Web malware*. Por isso, a proteção das aplicações *Web* também é um importante fator para deixar os usuários da Internet mais seguros. A seguir são expostas de maneira breve algumas técnicas que envolvem a análise de código e a detecção de anomalia para tornar as aplicações *Web* mais seguras.

Análise de código

Em [30, 48] são apresentadas técnicas de análise estática de código. O objetivo delas é traçar o fluxo de dados das aplicações, desde o ponto de entrada de dados até funções que podem sofrer abuso, verificando se existe algum mecanismo de validação neste caminho. Desta forma, pode-se determinar os pontos vulneráveis e de modo que seja possível tomar alguma providência para proteger estes pontos, como por exemplo pela validação da entrada de dados. Validação de entrada é uma técnica bastante utilizada e recomendada na proteção de aplicações *Web* mas, como qualquer mecanismo de proteção, pode conter erros. Em [3] é proposta uma ferramenta que analisa os métodos de validação usados e verifica se estes podem ser burlados por um atacante.

Detecção de anomalias

Além da análise de código, existem metodologias para detectar e prevenir ataques a aplicações *Web* enquanto estas estão sendo executadas. Em [45], Valeur et al. descrevem uma técnica para a prevenção de ataques utilizando um sistema detector de anomalias e um *proxy* reverso. Se o tráfego for considerado suspeito, é redirecionado para uma cópia da aplicação que não possui dados sensíveis.

3.3 Conclusão

Neste capítulo foram apresentadas as principais técnicas e sistemas utilizados para proteger os usuários da Internet durante a navegação. As técnicas descritas foram separadas de acordo com o local onde são implementadas. Elas podem ser implementadas diretamente no lado cliente, sem que sua segurança dependa de operações que necessitem ser configuradas no lado servidor. Também podem ser implementadas com técnicas conjuntas que atuam no lado cliente e servidor, aproveitando informações sobre a estrutura de cada aplicação *Web*. E por último, podem ser implementadas apenas no lado servidor, como forma de impedir que páginas legítimas sejam comprometidas para hospedar códigos maliciosos.

Capítulo 4

Sistema desenvolvido

Neste capítulo são apresentados o sistema de análise de *Web malware* desenvolvido, chamado de BroAD (*Browser Attacks Detection*), sua arquitetura, a descrição dos componentes, a forma como o sistema é configurado, o processo de análise aplicado e os métodos de detecção utilizados.

4.1 Arquitetura e componentes

Nesta seção são apresentados em detalhes os componentes do sistema desenvolvido, as interações entre eles e o papel de cada um no processo de análise. O sistema BroAD analisa dinamicamente amostras de código malicioso em um ambiente virtual. O uso deste tipo de ambiente permite que o sistema de análise seja restaurado após seu comprometimento. Neste contexto, o sistema *host* é o sistema operacional instalado na máquina base, enquanto que o sistema *guest* é o sistema operacional executado dentro do ambiente virtualizado.

Para determinar se uma página contém código malicioso ou não, primeiramente é feita a captura do comportamento do JavaScript e as chamadas de sistema efetuadas pelo navegador *Web* enquanto acessa essa página. Estes dados são então analisados e o comportamento é classificado através da detecção em quatro etapas. Estas etapas são descritas mais à frente, na Seção 4.4.

Os componentes de BroAD estão organizados como a seguir. No sistema *host* há um programa de gerenciamento (analisador) cuja função é controlar o processo de análise, os programas que processam os dados capturados durante esta e identificar comportamentos maliciosos. A biblioteca responsável por capturar as ações do código JavaScript está localizada no sistema *guest*, assim como o *driver* de *kernel* que captura as chamadas de sistema e o programa de controle (controlador) que inicializa a biblioteca e o *driver*. O único dado que é capturado no sistema *host* é o tráfego de rede, que é monitorado com o

uso da ferramenta *tcpdump*. Esta arquitetura pode ser vista na Figura 4.1.

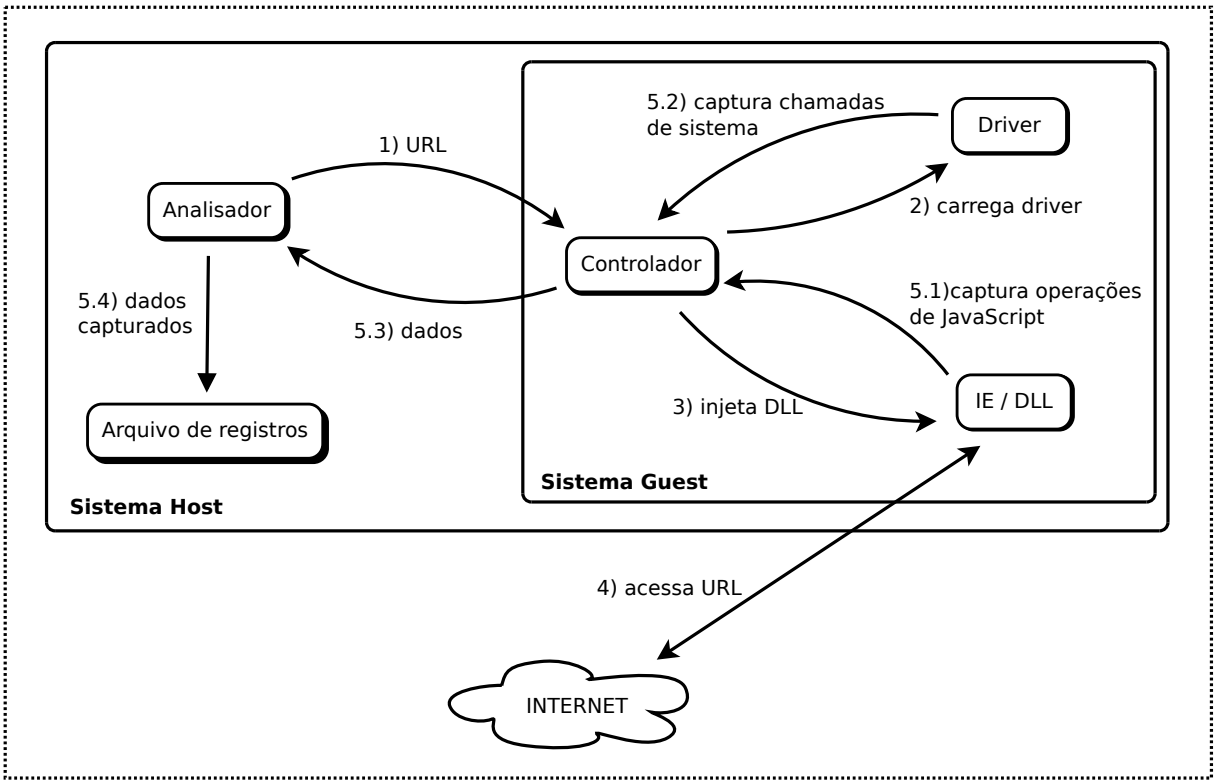

Figura 4.1: Arquitetura e componentes do sistema BroAD

O processo de análise é iniciado pelo componente analisador, o qual passa a URL a ser monitorada para o componente controlador. O controlador, por sua vez, carrega o *driver*, injeta o monitor de JavaScript na memória do Internet Explorer e passa a URL para ele. Enquanto o Internet Explorer acessa a URL passada, o comportamento do JavaScript e as chamadas de sistema capturados são encaminhados ao analisador, que os armazena no sistema *host*. O analisador deixa a análise ser executada durante 14 minutos. Este tempo relativamente alto é utilizado devido ao *overhead* causado pela operação do monitor em conjunto com o *driver*. Para tornar as análises mais rápidas, o monitor de JavaScript verifica se o navegador já terminou de processar a página e, caso isso aconteça antes do tempo previsto, a análise é finalizada antes que ocorra o *timeout* dos 14 minutos. Outra forma de realizar a análise de um conjunto de amostras de *Web malware* mais rapidamente é o uso de várias instâncias do sistema virtual ao mesmo tempo. A otimização do processo de análise para torná-lo mais rápido foi deixada como trabalho futuro, haja vista que foge do escopo desta dissertação.

4.1.1 Monitor de JavaScript

A monitoração de código JavaScript foi escolhida porque, atualmente, esta é a principal linguagem utilizada nos ataques [15]. Outras linguagens, como VBScript, são utilizadas, mas com menor frequência do que JavaScript. O monitoramento de outras linguagens é um dos trabalhos futuros.

Para monitorar o comportamento do código JavaScript é usada uma DLL de Windows. Esta biblioteca é carregada na memória do Internet Explorer pelo controlador e modifica outras bibliotecas do Internet Explorer, com a finalidade de alterar o fluxo de execução de determinadas funções, fazendo com que as funções do monitor de JavaScript (que registram as ações) sejam chamadas antes das funções originais. Este tipo de modificação é conhecido como *hook* e a metodologia usada neste trabalho para instalá-los foi baseada no *Ultimate Deobfuscator* [7].

Para determinar as posições onde os *hooks* são instalados é preciso manualmente analisar as bibliotecas do Internet Explorer e encontrar a localização dos endereços de memória nos quais os dados necessários podem ser obtidos. A DLL é então gerada com esses endereços específicos armazenados internamente ao código binário. Os *hooks* são normalmente instalados no início das funções e funcionam da seguinte maneira: primeiramente, as instruções iniciais após o ponto onde ocorreu a interceptação são copiadas para uma área de memória separada, chamada de trampolim. Na função original, estas instruções são substituídas por uma instrução *assembly JMP*[1] que redireciona o fluxo de execução para a área de trampolim. Nesta área, após as instruções copiadas, há outra instrução *JMP* para mudar o fluxo de execução para a função que grava os dados necessários, que normalmente são os parâmetros passados para a função original. A situação final da função interceptada pode ser vista na Figura 4.2.

Os endereços de memória onde os *hooks* são instalados, local onde as interceptações são feitas, são pré-definidos. Se o monitor de JavaScript desenvolvido precisar ser utilizado com outras versões do Internet Explorer e suas bibliotecas, é necessário modificar os endereços onde as interceptações são colocadas e a forma como os dados são capturados, caso os parâmetros das funções monitoradas ou estruturas internas das bibliotecas monitoradas mudem.

Um programa que possua acesso ao código em memória pode detectar o tipo de interceptação usado, mas os códigos interpretados pelo navegador que são carregados ao acessar alguma página *Web* não conseguem fazer acesso direto à memória, devido ao modelo de segurança utilizado nos navegadores. Logo, estes códigos maliciosos poderiam fazer a detecção apenas após o abuso de alguma vulnerabilidade no lado do cliente. Porém, para alcançar isto, o código em questão já teria que ter executado suas ações maliciosas, as

[1] A instrução *JMP* é um *jump* incondicional que modifica o fluxo de execução do programa.

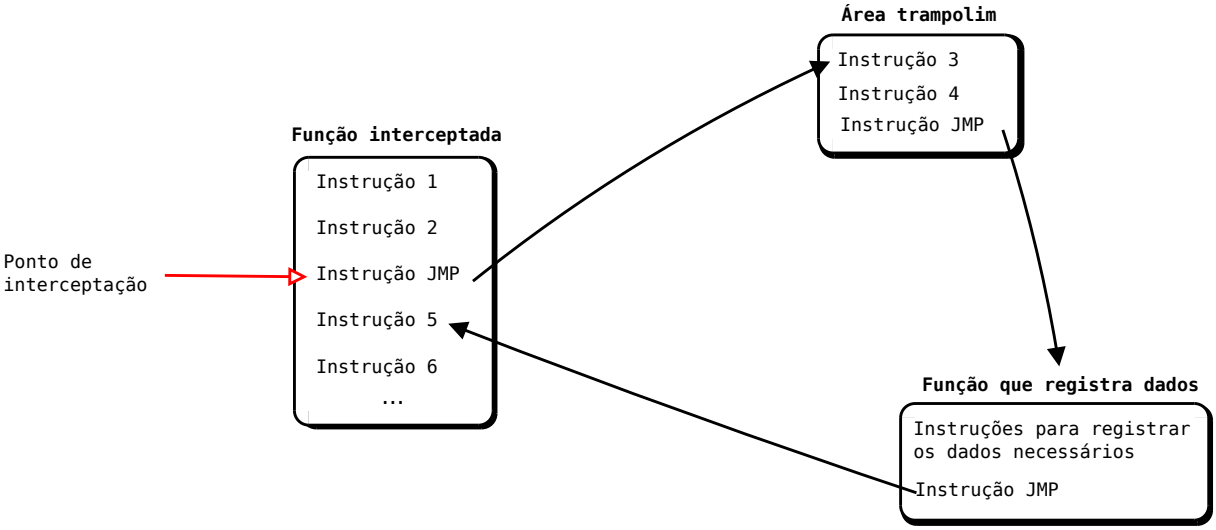

Figura 4.2: Situação da função interceptada após a instalação de um *hook*

quais, caso sejam programadas em JavaScript, seriam detectadas pelo sistema de análise.

Ações capturadas

As bibliotecas do Internet Explorer cujas funções internas são interceptadas são a *jscript.dll* (interpretação de JavaScript), a *mshtml.dll* (relacionada à estrutura da página *Web* apresentada no navegador) e a *msxml3.dll* (responsável por operações HTTP feitas através do código JavaScript). A Tabela 4.1 mostra as funções monitoradas que se referem a cada tipo de operação, enquanto que os tipos de ações monitoradas são apresentados e justificados a seguir.

Interpretador de JavaScript. São capturados todos os blocos de código JavaScript passados para o interpretador localizado na biblioteca *jscript.dll*.

Manipulação de *strings*. As funções de manipulação de *strings* são muito usadas para ofuscar o código. Ao capturar o resultado das chamadas a essas funções é possível identificar trechos que contêm código desofuscado.

Manipulação de vetores. Os vetores são muito usados em códigos que fazem *heap-spraying* (explicado na Seção 2.4.1). Assim, a monitoração de dados inseridos nos vetores é útil na detecção de *drive-by downloads* através da detecção de anomalias no comportamento do JavaScript e na detecção de *shellcodes*.

Modificações do DOM. As chamadas a funções de manipulação do DOM, os valores inseridos nas propriedades *innerHTML* e os valores inseridos em propriedades de elementos, como a propriedade *src* de um elemento *iframe*, são úteis na identificação de código com ofuscação e trechos desofuscados deste.

Propriedades do navegador. A monitoração de chamadas a funções que lêem propriedades do navegador, como *cookies* e cor de *links*, é importante na detecção de ataques de roubo de informações pessoais.

Manipulação de objetos ActiveX. Normalmente o abuso do Internet Explorer envolve vulnerabilidades de algum componente ActiveX, por isso a monitoração da criação destes e chamadas de seus métodos é importante.

Decodificação. A função *unescape* é muito utilizada para decodificar *shellcodes* na memória do navegador.

Execução de código dinamicamente. Funções que executam código dinamicamente são usadas para ofuscar o código, por isso a monitoração desses códigos pode revelar trechos desofuscados.

Acesso a páginas. O monitoramento da função que abre outra página indica uma das formas que podem ser usadas para carregar conteúdo externo.

Requisições HTTP. O monitoramento das requisições HTTP efetuadas é útil para identificar ataques que roubam informações pessoais e as enviam para outro domínio, este diferente do que está sendo acessado durante a análise.

Página carregada A monitoração desta função é importante para detectar quando o navegador terminou de carregar a página. Assim, a análise pode terminar antes do tempo limite.

Operação	**Funções monitoradas**
Interpretador de JavaScript	*ParseScriptTextCore*
Manipulação de *strings*	*CharAt, CharCodeAt, FromCharCode, StrReplace, StrSlice, StrSplit, SubStr* e resultado da concatenação de *strings*
Manipulação de vetores	*ArrJoin* e *ArrSetValue*
Modificações do DOM	*CloneNode, CreateElement, DocumentWrite, GetElementById, GetElementsByTagName, GetElementsByName, InsertElement, RemoveChild, PutinnerHTML* e *SetAttrib*
Propriedades do navegador	*ReadProp, SetProp, GetCookie, GetClipboard* e *GetColor*
Manipulação de objetos ActiveX	*SubInvoke* e criação de objetos ActiveX
Decodificação	*Unescape*
Execução de código dinamicamente	*Eval, SetTimeout*
Acesso a páginas	*Window.Open*
Requisições HTTP	*XMLHTTP.Open*
Página carregada	*LoadStatusDone*

Tabela 4.1: Funções monitoradas relacionadas a cada tipo de operação

Formato dos registros

Ao registrar as ações, o monitor de JavaScript utiliza a hora em que a ação foi realizada, o endereço da página que está sendo executada e a ação propriamente dita. A ação inclui o tipo de operação e os argumentos passados. Ações podem indicar a chamada do interpretador de JavaScript da *jscript.dll*, a chamada de alguma função, a modificação de alguma propriedade, a consulta de alguma propriedade ou a criação de um objeto. Já os argumentos indicam, no caso das funções, qual função foi chamada e os parâmetros que foram passados a ela ou seu valor de retorno. No caso da leitura ou modificação de alguma propriedade, os argumentos indicam qual é essa propriedade e, quando há a criação de objetos, indicam quais objetos foram criados. A Figura 4.3 apresenta exemplos de ações registradas.

```
00:49:41.903 http://<IP>/index4.html CALL DOM.GETELEMENTSBYTAGNAME
00:49:41.951 http://<IP>/index4.html CALL DOM.DOCUMENT.WRITE <Iframe width=100
height=0 src=../a1/Ms06014.htm></iframe>
00:49:42.031 http://<IP>/index4.html SET PROPERTY 63606C20 ../a1/Ms06014.htm
00:49:42.111 http://<IP>/index4.html SET PROPERTY 63606964 0
00:49:42.171 http://<IP>/index4.html SET PROPERTY 636069C0 100
00:49:42.472 http://<IP>/index4.html CREATE ACTIVEX Downloader.DLoader.1
```

Figura 4.3: Exemplos de ações registradas pelo monitor de JavaScript

4.1.2 *Driver* de *kernel*

O monitoramento de chamadas de sistema no nível do *kernel* é alcançado com o uso de um *driver* de *kernel* de Windows que modifica a tabela SSDT (*System Service Dispatch Table*). A SSDT é uma estrutura de *kernel* na qual são armazenados os endereços de memória das APIs nativas do sistema. Após ser carregado pelo controlador do BroAD, o *driver* modifica certos endereços da SSDT, substituindo-os por endereços de funções do próprio *driver*. Desta forma, garante-se que as funções do *driver* sejam chamadas antes das funções nativas que estão sendo monitoradas, registrando os dados necessários para posteriormente realizar a detecção de comportamentos maliciosos.

O *driver* é utilizado para capturar as chamadas de sistema realizadas pelo Internet Explorer e por qualquer processo que seja inicializado por ele. São capturadas as chamadas de sistema relacionadas a modificações de arquivos, modificações de registros, escritas em memória, carregamento de *drivers* e criação de processos. Além disso, são capturados os parâmetros das chamadas de sistema, os quais incluem, por exemplo, o nome do arquivo criado por uma chamada do tipo *CreateFile*.

O *driver* pode ser evadido ou detectado por certos *rootkits* (um tipo especial de *malware* que opera silenciosamente no nível do sistema operacional). Entretanto, para instalar tais

rootkits o *Web malware* teria que previamente abusar de alguma vulnerabilidade no navegador ou em outro componente. Se isto for feito com código JavaScript, o comportamento é registrado, o sistema então detecta tal comportamento e o classifica como malicioso, mesmo que as ações posteriores não sejam detectadas e monitoradas.

Este componente (*driver*) foi desenvolvido por outro aluno como parte de seu trabalho de mestrado e é explicado mais detalhadamente em [24].

4.1.3 Controlador

O componente controlador é responsável por iniciar a análise do *Web malware* assim que receber um comando do componente analisador. Após tal comando, o controlador injeta a DLL que monitora JavaScript na memória do Internet Explorer, carrega o *driver* que monitora as chamadas de sistema e passa os dados capturados para o componente analisador, que se situa fora do ambiente de análise.

4.1.4 Analisador

O componente analisador é responsável por inicializar o sistema virtualizado, passar a URL a ser analisada para o componente controlador, receber e armazenar os dados capturados e realizar a classificação do código analisado.

4.2 Configuração do sistema

O sistema operacional usado como *guest* é um Windows XP SP2, cujo navegador padrão instalado é o Internet Explorer 8 com componentes adicionais instalados que podem sofrer abuso por *Web malware*, tais como Adobe Reader, JRE (Java Runtime Environment) e Flash player.

Os exemplares de *Web malware* podem abusar de diversas aplicações diferentes instaladas no sistema dos usuários. Para poder analisar os diversos códigos maliciosos, os sistemas de análise precisam emular estas aplicações (nos casos de *honeyclients* de baixa interatividade) ou tê-las instaladas no sistema de análise (nos *honeyclients* de alta interatividade). Alguns *honeyclients* de baixa interatividade, como o PhoneyC, emulam apenas determinados objetos ActiveX vulneráveis, enquanto outros, como o JSand, emulam qualquer objeto requisitado pelo código malicioso.

Como descrito anteriormente, a emulação de cada objeto ActiveX requisitado ajuda na análise de códigos que abusam de diferentes aplicações, mas deixa o sistema facilmente detectável e evasível. Este fator foi decisivo no desenvolvimento do sistema BroAD, por isso escolhemos desenvolvê-lo como um sistema de alta interatividade. Por não emular objetos

ActiveX, o sistema não pode ser detectado pela técnica que cria um objeto inexistente, mas há um problema de flexibilidade: é preciso atualizar constantemente as aplicações instaladas no sistema *guest* para poder analisar os códigos maliciosos mais recentes. Sem essas aplicações, os códigos que as atacam podem não ser detectados. Por outro lado, caso o código carregue o *shellcode* na memória ou faça *heap-spraying* antes de verificar se o componente vulnerável está presente no sistema, o código malicioso poderá ser detectado. Devido a isto, optou-se por um custo maior de manutenção para garantir altas taxas de detecção e identificação dos códigos maliciosos, aumentando assim o nível de informações e alertas que podem ser providos por BroAD.

4.3 Informações providas

BroAD provê informações sobre o comportamento do JavaScript, as chamadas de sistema que foram consideradas anômalas e o tráfego de rede produzido durante a execução do *Web malware*. A informação relacionada à execução de JavaScript inclui os códigos que foram passados para o interpretador de JavaScript do Internet Explorer, os códigos passados para funções que os executam dinamicamente, as modificações no DOM, os *shellcodes* identificados, os objetos ActiveX usados e informações acerca do tráfego de rede, as quais incluem os endereços IP acessados e as requisições HTTP e DNS realizadas.

Entre os sistemas existentes que foram utilizados para avaliar a qualidade dos resultados produzidos pelo sistema proposto, JSand/Wepawet é o que provê mais informações sobre os exemplares de *Web malware* analisados. Entretanto, JSand não mostra as modificações feitas no DOM através de mudanças na propriedade *innerHTML*. Quando JSand/Wepawet detecta um executável malicioso, este provê um *link* para o relatório de análise da amostra no sistema Anubis, uma plataforma para analisar executáveis de Windows. Entretanto, se um *shellcode* utilizado em um ataque executa mais do que apenas o *download* e execução de um *malware*, o sistema JSand/Wepawet não será capaz de reportar as ações realizadas. Os outros sistemas avaliados provêem ainda menos informações; CaptureHPC apenas informa as chamadas de sistema que foram consideradas anômalas e PhoneyC provê somente informações sobre *exploits* e objetos ActiveX usados.

4.4 Detecção

A detecção de comportamento malicioso é realizada pela técnica de quatro etapas desenvolvida neste trabalho—verificação de assinaturas de JavaScript, verificação de *shellcodes*, análise de chamadas de sistema e a detecção de anomalia usando oito atributos extraídos das ações de JavaScript. Estas etapas são detalhadas nas próximas seções.

4.4.1 Assinaturas de JavaScript

As assinaturas de JavaScript que foram desenvolvidas são utilizadas para identificar padrões de ataques conhecidos. As assinaturas são formadas por expressões regulares cujo propósito é verificar certas ações e seus parâmetros associados. Cada assinatura possui um número de identificação, um texto descritivo, um atributo de ordenação que indica se as ações precisam ser encontradas na ordem apresentada para a assinatura ser marcada como encontrada, e uma quantidade variável de ações. Cada ação é formada por uma operação, um número inteiro indicando a quantidade de parâmetros a serem verificados e os próprios parâmetros (tantos quantos tiverem sido indicados). A operação e os parâmetros são expressões regulares que verificam, respectivamente, a função registrada e seus argumentos. A Figura 4.4 mostra a estrutura descrita.

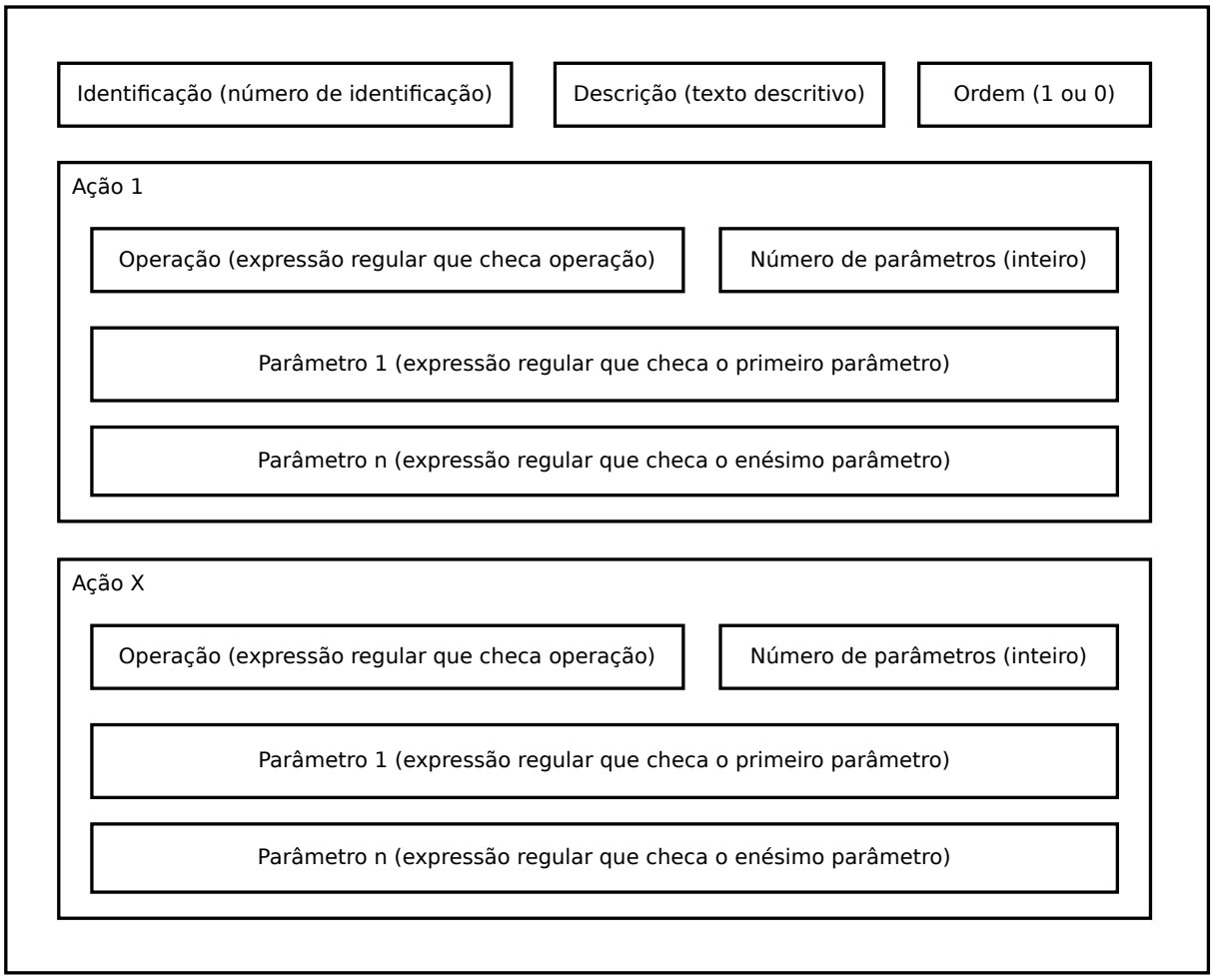

Figura 4.4: Estrutura de uma assinatura de JavaScript

As assinaturas de JavaScript são usadas principalmente para detectar ataques de roubo

de informações, que precisam seguir uma determinada sequência de ações para serem executados. Um exemplo de sequência de ações para um código que verifica se o usuário visitou o *Web site www.teste.com* é apresentado a seguir:

- Criação de um link (*tag* 'a') apontando (propriedade *src*) para o *site www.teste.com*;
- Inserção do *link* criado na página;
- Obtenção da cor do *link*;
- Verificação, através da cor do *link*, sobre a possibilidade de saber se o usuário já visitou o *site*. Em caso positivo, ocorre o envio da informação para *www.malicioso.com*.

Na Tabela 4.2 é apresentada a assinatura usada para detectar esse ataque. O parâmetro utilizado com a operação de envio de informações realiza a checagem se foi feita uma requisição HTTP para um domínio diferente do qual foi obtido o código. A expressão $\$\{SOURCE\}$ é substituída pelo nome de domínio sob análise. É importante notar que esta não é a única maneira de se realizar o ataque, principalmente porque existem diversas formas de enviar a informação para um *site*. Isto implica que é necessário criar assinaturas para todos os casos, como em todo sistema baseado em assinaturas.

Operação	**Regex da operação**	**Regex do parâmetro**
Cria *link*	CALL DOM.CREATE_ELEMENT	^[aA]$
Insere *link*	CALL DOM.INSERTELEMENT	-
Pega cor	GET COLOR	-
Envia informação	CALL XMLHTTP.OPEN	^(?!${SOURCE}).*$

Tabela 4.2: Expressões regulares usadas para detectar um ataque de roubo de histórico de navegação

As assinaturas de JavaScript também podem ser usadas para identificar os *exploits* usados por códigos maliciosos. Esta funcionalidade não foi implementada em BroAD, deixada como trabalho futuro. Entretanto, se o classificador identifica uma página *Web* como maliciosa, as assinaturas de JavaScript podem ser facilmente aplicadas para identificar qual vulnerabilidade sofreu abuso, baseando-se nas interações com objetos ActiveX.

4.4.2 Verificação de *shellcodes*

Como descrito na Seção 2.4.1, após o comprometimento do navegador ou de algum de seus componentes, é executado um *shellcode*. A partir desta execução, o atacante pode comprometer o sistema da vítima através da instalação de algum *malware* obtido após o abuso ter sucesso. Como os *shellcodes* são normalmente desenvolvidos em código de

máquina da arquitetura x86, é utilizada a biblioteca libemu [2] para verificar a existência deles. Esta biblioteca verifica se um determinado conjunto de dados forma um conjunto válido de instruções x86.

As *strings* que são usadas como parâmetro, ou as que são resultado de certas funções monitoradas de JavaScript, são examinadas e seu *mime-type* é verificado. O *mime-type* indica o tipo de conteúdo encontrado. Se a *string* é considerada suspeita, ou seja, se o *mime-type* não contém o termo "*text*", esta é verificada com a libemu. Caso seja identificado um conjunto válido de instruções, considera-se que um *shellcode* foi detectado e a página analisada é classificada como maliciosa, dado que a existência de código de máquina não é considerada normal em códigos JavaScript.

A Figura 4.5 mostra um exemplo da maneira que um *shellcode* é carregado em uma variável de JavaScript. A função *unescape* é usada para decodificar *strings* e seu resultado é monitorado pela DLL. Neste caso, o *mime-type* do conteúdo retornado na chamada é do tipo "*application/octet-stream*" e, quando a *string* é verificada pela libemu, é marcada como contendo código de máquina.

```
var shellcode = unescape("%uC033%u8B64%u3040%u0C78%u408B%u8B0C%u1C70%u8BAD%u0858%u09EB%u408B%
u8D34%u7C40%u588B%u6A3C%u5A44%uE2D1%uE22B%uEC8B%u4FEB%u525A%uEA83%u8956%u0455%u5756%u738B%u8B3C%
u3374%u0378%u56F3%u768B%u0320%u33F3%u49C9%u4150%u33AD%u36FF%uBE0F%u0314%uF238%u0874%uCFC1%u030D%
u40FA%uEFEB%u3B58%u75F8%u5EE5%u468B%u0324%u66C3%u0C8B%u8B48%u1C56%uD303%u048B%u038A%u5FC3%u505E%
u8DC3%u087D%u5257%u33B8%u8ACA%uE85B%uFFA2%uFFFF%uC032%uF78B%uAEF2%uB84F%u2E65%u7865%u66AB%u6698%
uB0AB%u8A6C%u98E0%u6850%u6E6F%u642E%u7568%u6C72%u546D%u8EB8%u0E4E%uFFEC%u0455%u5093%uC033%u5050%
u8B56%u0455%uC283%u837F%u31C2%u5052%u36B8%u2F1A%uFF70%u0455%u335B%u57FF%uB856%uFE98%u0E8A%u55FF%
u5704%uEFB8%uE0CE%uFF60%u0455%u7468%u7074%u2F3A%u6B2F%u7265%u6972%u636D%u656B%u7465%u2E71%u6E69%
u6F66%u732F%u6C2F%u616F%u2E64%u6870%u3F70%u3D65%u0032");
```

Figura 4.5: *Shellcode* sendo decodificado e carregado em uma variável de JavaScript

A libemu também é usada para gerar um gráfico que representa o fluxo de execução do código *assembly* detectado. A Figura 4.6 apresenta o gráfico gerado a partir do *shellcode* apresentado na Figura 4.5.

O conjunto de instruções x86 é bastante denso, isto é, muitos conjuntos de *bytes* formam um grupo de instruções válidas. Entretanto, em JavaScript, *strings* que contêm texto são codificadas com Unicode de 16 *bits*, fazendo com que uma sequência de caracteres ASCII, por exemplo, contenha um *byte* 0x0 a cada dois outros *bytes* [15].

A verificação de *shellcodes* também é uma forma de se fazer detecção por assinatura, mas por não utilizar o mesmo formato das assinaturas apresentadas na Seção 4.4.1, decidiu-se colocá-la em uma etapa separada.

4.4.3 Verificação de chamadas de sistema

As chamadas de sistema são verificadas procurando-se por ações executadas no sistema operacional que não deveriam ser realizadas sem o consentimento do usuário. Como a

análise de BroAD é executada automaticamente, sem nenhuma interação humana, todas as chamadas de sistema executadas são suspeitas, pois ocorrem por intermédio do uso do navegador quando acessando uma página analisada. Se, por exemplo, um processo é iniciado a partir do diretório de arquivos temporários do Internet Explorer, este é considerado malicioso.

Esta verificação é feita com o uso de expressões regulares cujo propósito é definir os conjuntos de chamadas de sistema que são aceitos como inofensivos e quais delas configuram comportamento malicioso. Assim, as expressões regulares podem detectar ataques bem sucedidos que resultem na instalação de *malware* ou que resultem em alguma modificação no sistema de arquivos, a qual não deveria ser executada automaticamente pelo navegador em condições normais. Esta verificação é importante principalmente em casos que não utilizam a linguagem JavaScript na fase de abuso, como em ataques via Java, VBScript ou Flash. A Figura 4.7 apresenta uma expressão regular que permite a manipulação de arquivos no diretório onde são armazenados *cookies*. Esta permissão é necessária porque o Internet Explorer faz esse tipo de operação automaticamente, ao acessar qualquer página *Web* que utilize *cookies*.

4.4.4 Anomalia

Para realizar a detecção por anomalia são usados oito atributos extraídos das ações de JavaScript. Destes, cinco são provenientes dos atributos de número 3 a 8 presentes em [9]. Os demais são propostos no âmbito deste trabalho, sendo que foi adicionado um atributo que representa a soma do tamanho das *strings* que foram identificadas como possíveis *shellcodes* e outro no qual são consideradas as alocações de *strings* em vetores. Como mencionado anteriormente, as *strings* consideradas suspeitas são aquelas cujo *mime-type* não é do tipo *text*. Os atributos utilizados na detecção por anomalia de BroAD são apresentados na Tabela 4.3 e detalhados a seguir.

Definições de *strings* e alocações em vetores. Códigos JavaScript ofuscados comumente realizam muitas operações de *string* antes de apresentarem o comportamento malicioso e, quando lançam ataques do tipo *heap-spraying*, normalmente alocam uma grande quantidade de *strings* com tamanhos também grandes em vetores.

Código executado dinamicamente. Códigos ofuscados também costumam executar dinamicamente trechos desofuscados.

Modificações no DOM. A modificação no DOM é uma outra característica geralmente encontrada em códigos ofuscados.

Possíveis *shellcodes*. A presença destes pode ser um indicativo da ocorrência de uma tentativa de ataque de *heap-spraying*.

Objetos ActiveX. Para aumentar a chance de abuso das vítimas, muitos códigos

tentam abusar de vários componentes ActiveX; além disso, este abuso pode envolver a passagem de longos parâmetros para algum dos métodos deste tipo de objeto.

1	Número de definições de *strings* e alocações de *strings* em vetores
2	Tamanho de *strings* definidas e *strings* alocadas em vetores
3	Número de chamadas a funções que executam código dinamicamente e funções que modificam o DOM
4	Tamanho de trechos de código executados dinamicamente
5	Número de possíveis *shellcodes* encontrados
6	Tamanho de possíveis *shellcodes* encontrados
7	Número de objetos ActiveX criados
8	Tamanho dos parâmetros passados para funções presentes nos objetos ActiveX

Tabela 4.3: Atributos utilizados na detecção por anomalia

Ataques por *drive-by download* normalmente utilizam a linguagem JavaScript para carregar o *shellcode* na memória e ativar a vulnerabilidade presente no navegador ou em algum de seus componentes. Mesmo que este tipo de ataque não seja inteiramente bem-sucedido, ainda é possível detectá-lo devido aos atributos extraídos na tentativa de comprometimento, caso pelo menos a etapa de carregar o *shellcode* na memória seja completada. Isto acontece devido à alocação de várias instâncias do *shellcode* na memória, como parte do processo de *heap-spraying*, permitindo sua detecção por BroAD.

Para criar o classificador foi usada a plataforma Weka [27], da qual foram escolhidos o meta-classificador *ThresholdSelection* e o algoritmo classificador *RandomForest* [4], que cria diversas árvores de decisão com seleção aleatória de atributos e então elege a classe que foi selecionada por mais árvores. Estes algoritmos foram escolhidos por apresentarem os melhores resultados em testes preliminares realizados. No próximo capítulo, serão apresentados os testes realizados e os resultados obtidos com a aplicação dos classificadores produzidos em milhares de amostras de *Web malware*.

4.5 Conclusão

Neste capítulo foi apresentado o sistema desenvolvido, BroAd, que faz a monitoração das ações do JavaScript e das chamadas de sistema, e que é capaz de classificar as páginas *Web* analisadas como maliciosas ou benignas. A detecção do comportamento malicioso é feita através de quatro etapas, que foram detalhadas uma a uma. Estas etapas são a verificação de assinaturas de JavaScript, a verificação por *shellcodes*, a verificação de chamadas de sistema e a detecção de anomalia no comportamento do código JavaScript. Além disso, foram apresentadas as informações providas pelo sistema proposto sobre os códigos analisados.

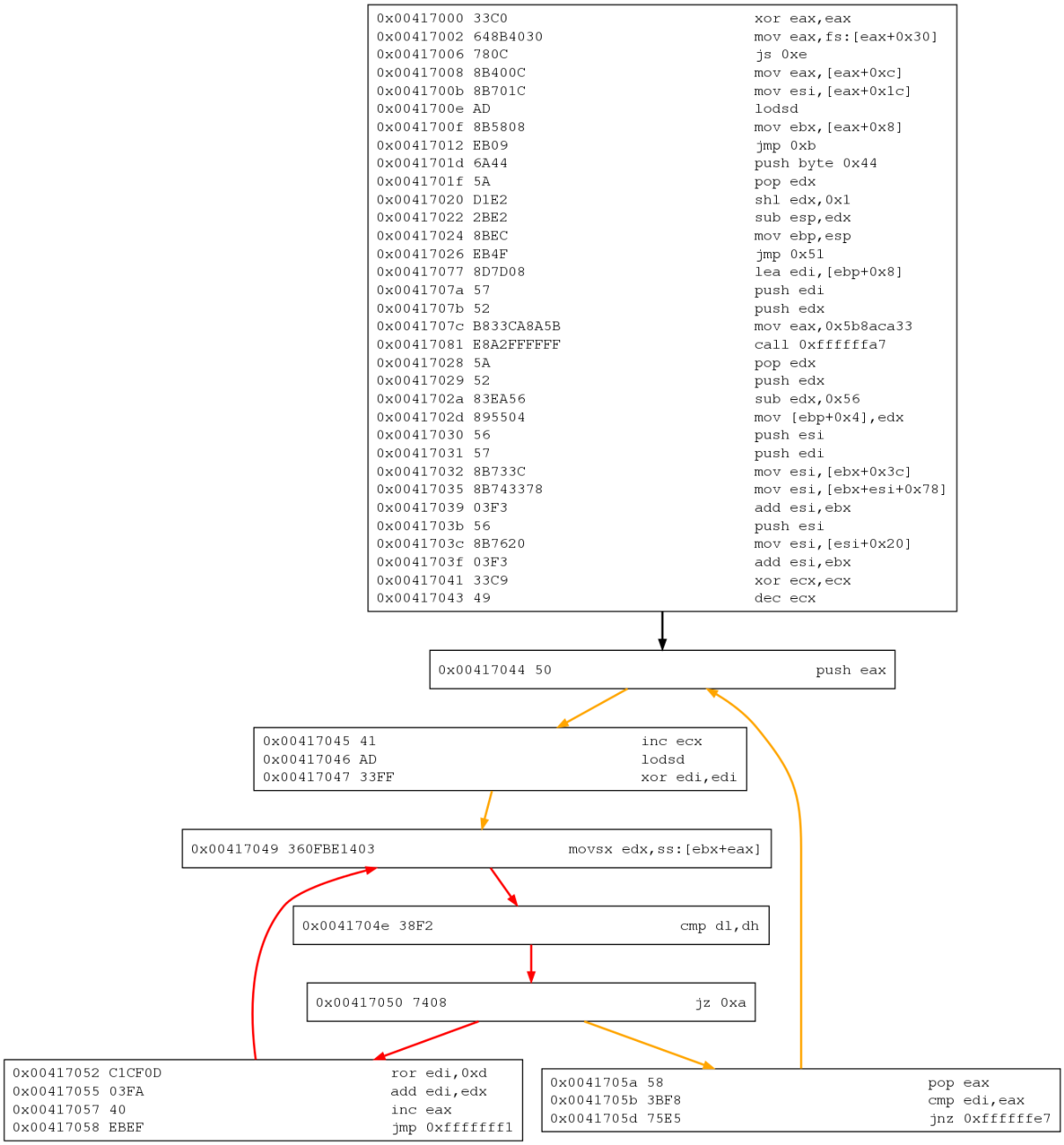

Figura 4.6: Fluxo de execução das instruções traduzidas pela libemu

```
.*;C:\\Program Files\\Internet Explorer\\iexplore.exe;(Create|Delete|Open|Write)File;C:\
\Documents and Settings\\Administrator\\Cookies\\.*
```

Figura 4.7: Exemplo de expressão regular que permite certas chamadas de sistema que manipulam arquivos do diretório que contém *cookies*

Capítulo 5

Testes e Resultados

Neste capítulo são apresentados os testes realizados para demonstrar a eficácia do sistema em comparação com três sistemas do estado-da-arte de análise de *Web malware*. São mostradas as taxas de detecção, as diferenças entre os tipos das informações que podem ser providas e a abrangência da análise de cada sistema. Além disso, é feita uma análise acerca de como cada etapa de detecção do sistema BroAD contribuiu para o resultado apresentado.

5.1 Conjuntos de dados e sistemas comparados

Para mostrar a eficácia do sistema BroAD, foi feita uma comparação com os três sistemas de análise mais populares atualmente (e publicamente disponíveis)—JSand, PhoneyC e Capture-HPC. Foram utilizados dois conjuntos de dados compostos por arquivos HTML e URLs, um deles com arquivos HTML sabidamente maliciosos e outro de URLs categorizadas como benignas. O conjunto de amostras maliciosas foi obtido de duas fontes: o pacote de amostras de códigos maliciosos do VXHeaven[1], o qual possui uma grande variedade de tipos de arquivos, incluindo arquivos executáveis, páginas HTML, arquivos compactados, e de páginas listadas pelo *Malware Domain List*[2], cujo *site* lista páginas que foram reportadas por grupos de pesquisa como hospedeiras de código malicioso. O conjunto, no total, contém 2389 amostras e apenas ataques do tipo *drive-by download*, por este ser o tipo mais comum em atividade. Já o conjunto de amostras benignas foi obtido das 12 mil primeiras entradas do site Alexa[3], que lista as páginas mais acessadas da Internet. Além disso, as URLs consideradas benignas foram verificadas com o serviço

[1] http://vx.netlux.org
[2] http://www.malwaredomainlist.com/
[3] http://www.alexa.com/

de *safe browsing* do Google[4] e as amostras que foram apontadas por conter conteúdo malicioso foram removidas do conjunto, resultando em 10.096 amostras.

Cada um dos conjuntos (maligno e benigno) foi separado em dois subconjuntos, um para treinamento e outro para testes de validação. Os conjuntos de treinamento e de testes das amostras maliciosas contêm 717 e 1.672 amostras, respectivamente. Já os conjuntos de treinamento e teste das amostras benignas contém 3.077 e 7.019 amostras, respectivamente. Assim, o conjunto total de amostras para treinamento contém 3.794 e o de testes contém 8.691 amostras no total, entre maliciosas e benignas. Os conjuntos de treinamento foram utilizados para treinar o classificador usado na detecção por anomalia.

5.2 Resultados

A validação utilizando a separação dos dados de treinamento em dez amostras (10-*fold*) resultou em 0,91% de falso-positivos— exemplares benignos classificados como maliciosos—e 18,41% de falso-negativos—exemplares maliciosos não detectados. O resultado da detecção utilizando os conjuntos de dados mencionados e os quatro sistemas (BroAD e os outros três mencionados) são apresentados na Tabela 5.1. Os sistemas BroAD, PhoneyC e Capture-HPC classificam as amostras como maliciosas ou benignas, enquanto o JSand possui uma terceira classificação, os suspeitos. As amostras que estão na coluna de erro foram classificadas assim devido a problemas na execução do sistema. Nestes casos, o sistema apresentou algum erro no momento da análise, ou por não ser capaz de analisar a amostra ou por problemas de implementação.

Conjunto	**BENIGNOS**				**MALICIOSOS**			
Detectado como	**M**	**S**	**B**	**E**	**M**	**S**	**B**	**E**
BroAD	22	0	6997	0	1356	0	316	0
JSand	1	126	5881	1011	335	116	1211	10
PhoneyC	0	0	5943	1076	180	0	1368	124
Capture-HPC	10	0	6117	892	96	0	1522	54

Tabela 5.1: Resultado da classificação de cada sistema, onde M = malicioso, S = suspeito, B = benigno e E = erro.

A Tabela 5.2 mostra a quantidade de falso-positivos, falso-negativos, verdadeiro-positivos (amostras maliciosas classificadas corretamente) e verdadeiro-negativos (amostras benignas classificadas corretamente) produzidos por cada sistema. Para calcular estes valores, as amostras classificadas pelo sistema JSand como suspeitas foram agrupadas com as consideradas maliciosas, a fim de se ter um resultado consistente.

[4]http://code.google.com/apis/safebrowsing/

Sistema	**FP(%)**	**FN(%)**	**VP(%)**	**VN(%)**
BroAD	0,31	18,90	81,10	99,69
JSand	1,81	72,43	26,97	83,79
PhoneyC	0	81,82	10,77	84,67
CaptureHPC	0,14	91,03	5,74	87,15

Tabela 5.2: Quantidade de falso-positivos (FP), falso-negativos (FN), verdadeiro-positivos (VP) e verdadeiro-negativos (VN) de cada um dos sistemas avaliados.

É uma tarefa difícil comparar o resultado dos sistemas utilizando apenas os dados apresentados na Tabela 5.2, pois é preciso considerar diversos fatores importantes ao mesmo tempo, como a quantidade de amostras maliciosas que foram detectadas corretamente e a quantidade de amostras benignas que foram classificadas como maliciosas. Por isso, foi utilizada a média harmônica (F-*measure*)[5] na comparação dos resultados providos pelos sistemas avaliados. A média harmônica considera tanto a precisão como o *recall* para calcular a qualidade dos resultados. A precisão indica a fração das amostras classificadas como maliciosas que realmente o são e é calculada por $P = \frac{VP}{(VP+FP)}$, enquanto o *recall* indica a fração das amostras previamente classificadas como maliciosas que foram classificadas assim também pelo sistema sob avaliação e é calculado por $R = \frac{VP}{(VP+FN)}$. Para calcular a média harmônica, é utilizada a fórmula F-*measure* $= \frac{2 \times R \times P}{R+P}$. A Tabela 5.3 mostra o valor dessas propriedades para cada sistema.

Sistema	***Recall*(%)**	**Precisão(%)**	**Média Harmônica(%)**
BroAD	81,1	99,62	89,41
JSand	27,13	93,71	42,08
PhoneyC	11,63	100	20,84
Capture-HPC	5,93	97,62	11,18

Tabela 5.3: Cálculo do *recall*, precisão e a média harmônica baseados nos resultados dos sistemas.

Pode-se notar que a média harmônica do BroAD é 89,41, mais do que o dobro do valor relativo ao segundo melhor sistema (JSand). Isto demostra a qualidade da abordagem proposta nesta dissertação, que mesmo apresentando uma precisão um pouco menor do que a obtida com o sistema PhoneyC, possui um valor de *recall* maior do que a soma do valor apresentado por todos os outros sistemas. A grande diferença no *recall* indica que o sistema BroAD é capaz de detectar mais amostras maliciosas do que os outros sistemas, já os valores próximos de precisão indicam que os sistemas possuem taxas de falso-positivos semelhantes.

[5]http://en.wikipedia.org/wiki/Precision_and_recall

5.3 Informações providas

A Tabela 5.4 mostra as informações sobre o comportamento do código analisado que são providas por cada sistema. Pode-se notar que o sistema Capture-HPC, por capturar apenas as chamadas de sistema, não é capaz de prover informações sobre o comportamento dos *scripts* executados em uma página analisada. O BroAD e o JSand são os que mais provêem informações a respeito do comportamento do código, com a diferença que o JSand pode apresentar quais vulnerabilidades sofreram abuso, enquanto o BroAD pode apresentar mais informações sobre o código desofuscado e modificações na estrutura da página. Neste trabalho, concluiu-se que as informações produzidas por BroAD são mais relevantes no escopo da segurança de clientes *Web*, dado que estão diretamente ligadas ao comportamento do código. Já as vulnerabilidades que sofreram abuso dizem mais a respeito do tipo de usuário (e aplicações) alvo de um ataque do que do comportamento do código malicioso propriamente dito, podendo ser obtidas mais facilmente na Internet.

Tipo de informação	**JSand**	**PhoneyC**	**Capture-HPC**	**BroAD**
Manipulação de objetos ActiveX	X	X		X
Vulnerabilidades que sofreram abuso	X	X		
Informações sobre *shellcodes* utilizados	X	X		X
Chamadas de sistema anômalas efetuadas			X	X
Informações sobre tráfego de rede	X			X
Código desofuscado por *eval*	X			X
Código desofuscado por *document.write*	X			X
Código desofuscado por *setTimeout*				X
Código desofuscado por *innerHTML*				X
Outras modificações no DOM				X

Tabela 5.4: Informações providas por cada um dos sistemas avaliados.

5.4 Abrangência da análise

A Tabela 5.5 mostra os tipos de ataque que cada sistema é capaz de detectar e avaliar, bem como as linguagens de programação usadas nestes ataques e que podem ser analisadas.

Por combinar a análise de chamadas de sistema com a monitoração do comportamento do código JavaScript, o sistema BroAD possui uma abrangência maior do que os outros sistemas usados na comparação.

Como o Capture-HPC e o BroAD fazem análise no nível do sistema operacional, podem detectar ataques independentemente da linguagem utilizada pelas páginas *Web* maliciosas, enquanto o JSand fica restrito a JavaScript e o PhoneyC a JavaScript e VBScript. A desvantagem da detecção por chamadas de sistema é o fato dela acontecer apenas quando o ataque é bem sucedido. Entretanto, como BroAD utiliza também a análise de JavaScript, pode detectar ataques nesta linguagem, em certos casos, mesmo quando estes não são bem sucedidos. Além disso, BroAD pode detectar ataques de roubo de informações do usuário codificados em JavaScript.

	Linguagem			**Tipo de ataque**	
Sistemas	**JS**	**VBS**	**Outras**	**DBD**	**Roubo de inf.**
BroAD	X	X	X	X	X (se usar JavaScript)
JSand	X			X	
PhoneyC	X	X		X	
Capture-HPC	X	X	X	X	

Tabela 5.5: Ataques (DBD - *drive-by download* - e roubo de informações) e linguagens (JS - JavaScript, VBS - Visual Basic Script) usadas nos ataques que cada sistema é capaz de analisar.

5.5 Etapas da detecção de BroAD

A Tabela 5.6 mostra quantas amostras cada método de detecção do sistema BroAD classificou como maliciosas, contribuindo para o total de verdadeiro-positivos e falso-positivos. Pode-se notar que as porcentagens não somam necessariamente 100% porque mais de um dos métodos empregados pode classificar as mesmas amostras como maliciosas.

Método de detecção	**VP**	**FP**
Anomalia	1279(94,32%)	20(90,91%)
Assinaturas de JavaScript	0(0%)	0(0%)
Detecção de *shellcodes*	29(2,14%)	0(0%)
Verificação de chamadas de sistema	276(20,35%)	2(9,09%)

Tabela 5.6: Técnica utilizada em cada método de detecção e como esta contribuiu para o total de verdadeiro-positivos e falso-positivos

As assinaturas de JavaScript não detectaram nenhuma amostra como maliciosa porque não haviam exemplares que realizavam roubo de informação nos conjuntos analisados.

A detecção de *shellcodes* não apresenta falso-positivos porque as *strings* de JavaScript, como mencionado na Seção 4.4.2, são codificadas com Unicode de 16 *bits*, por isso possuem um *byte* 0x0 em cada dois caracteres de um texto ASCII. Embora a falta de falso-positivos seja uma característica ideal, a quantidade de verdadeiro-positivos é baixa devido a limitações da libemu, a biblioteca usada para detectar os trechos de código de máquina presentes em *shellcodes*.

A detecção através das chamadas de sistema resultou em poucos falso-positivos e uma taxa também relativamente baixa de verdadeiro-positivos. Isto aconteceu, principalmente, porque grande parte das amostras utilizadas nos testes não resultou no comprometimento do sistema. Como a maioria das amostras presentes faz uso da linguagem JavaScript, a detecção por anomalia foi bastante eficaz. Entretanto, ao se analisar amostras maliciosas que utilizam outras linguagens de programação, como Java, a detecção por anomalia não funciona, tornando a monitoração das chamadas de sistema a única alternativa de se detectar o código como malicioso.

Como já mencionado, a maior parte das amostras se utiliza de JavaScript, que é a principal linguagem de *script* utilizada na *Web* [15], e por isso a detecção por anomalia do comportamento JavaScript foi bastante eficaz. Este tipo de detecção também resultou em uma quantidade maior de falso-positivos, porém, como mostrado na Tabela 5.2, esta quantidade pode ser considerada baixa, quando comparada com o valor produzido pelos outros sistemas avaliados.

Nos testes apresentados, a maior parte da detecção de verdadeiro-positivos foi feita através do uso da técnica de detecção por anomalia. Entretanto, as outras formas de detecção introduzem pouco tempo extra de processamento e possuem inúmeras vantagens, mesmo que não tenha sido o caso nos testes realizados. As assinaturas de JavaScript, que podem detectar ataques de roubo de informações, são uma vantagem sobre os outros sistemas, cujo foco é voltado para ataques de *drive-by download*. A verificação de *shellcodes* possui uma taxa de detecção relativamente baixa, mas não gerou nenhum falso-positivo. Por fim, a detecção por chamadas de sistema, que sozinha foi capaz de detectar mais amostras maliciosas do que o Capture-HPC, o qual também realiza detecção dessa forma, é o único meio de se detectar ataques (no sistema proposto) que não se utilizam de JavaScript.

Apesar do JavaScript ser a forma mais utilizada atualmente para realizar os ataques, o grande foco dado a essa linguagem pelos sistemas de análise e de proteção estão fazendo com que a quantidade de ataques que utilizam outras linguagens cresça. Dos sistemas utilizados na comparação, os únicos que são capazes de analisar amostras em diferentes linguagens são o Capture-HPC e o BroAD. O PhoneyC analisa também VBScript, mas fica restrito a esta e JavaScript.

5.6 Conclusão

Neste capítulo foram apresentados os resultados da comparação do sistema BroAD com os três principais sistemas de análise de *Web malware* existentes e disponibilizados publicamente. Estes testes demonstram que o sistema proposto possui uma taxa de detecção consideravelmente melhor do que os outros (mais do que o dobro do segundo colocado). Além disso, foram apresentadas as diferenças entre as informações apresentadas por cada sistema sobre as páginas *Web* avaliadas e a abrangência de cada sistema, mostrando que BroAD pode prover mais informações sobre o comportamento do código malicioso sob análise e possui uma abrangência maior em relação aos tipos de código que podem ser analisados e ataques que podem ser detectados. Por fim, foi apresentada a quantidade de verdadeiro-positivos e de falso-positivos gerados por cada tipo de detecção utilizado em BroAD, mostrando a relevância de cada um deles nos resultados dos testes e para a qualidade do sistema de forma geral.

Capítulo 6

Conclusões

A análise de *Web malware* é uma tarefa importante para o desenvolvimento de contramedidas e sistemas de proteção para navegadores *Web* modernos, uma vez que estes são alvos valiosos para atacantes que desejam instalar *malware* ou roubar informações dos usuários. Nesse trabalho foram apresentados os principais sistemas disponíveis, e considerados estado-da-arte, utilizados para realizar essa análise. Além disso, foram mostradas as limitações que esses sistemas possuem e códigos que podem detectar que estão sendo analisados por parte desses sistemas, evadindo esse processo e impedindo que eles provejam informações relevantes sobre o código.

Com a finalidade de corrigir tais limitações e preencher um espaço na área de análise, foi desenvolvida uma nova abordagem para detectar *Web malware*. O sistema proposto, chamado de BroAD (*Browser Attacks Detection*), combina a monitoração de chamadas de sistema e de comportamento do JavaScript com uma técnica que utiliza quatro etapas para realizar a detecção de atividades maliciosas, sendo elas a verificação de *shellcodes*, a detecção de anomalias no comportamento do código JavaScript, a verificação das chamadas de sistema e das assinaturas de comportamento de JavaScript. Este sistema foi desenvolvido utilizando o Windows XP. Para poder utilizá-lo em outras versões deste sistema operacional, como o Windows 7, seria necessário fazer modificações nos índices da SSDT usados pelo *driver* para acessar as funções da SSDT corretamente e modificações nos endereços interceptados pelo monitor de JavaScript.

Para demonstrar a efetividade e os melhores resultados produzidos pelo sistema proposto em relação aos sistemas existentes, foram feitos testes (treinamento e classificação) com mais de 12.000 URLs, previamente classificadas como benignas ou maliciosas, obtidas de diversas fontes: uma página *Web* que lista *sites* com conteúdo malicioso, um conjunto de códigos maliciosos disponível *online* e uma página *Web* que lista os *sites* mais acessados da Internet. Para comparar a qualidade da detecção dos sistemas avaliados foi utilizada a média harmônica, que por sua vez é calculada através da precisão e do *recall* referentes

à detecção realizada. A média harmônica do sistema desenvolvido, calculada a partir dos testes de treinamento e classificação, apresentou um resultado mais de duas vezes melhor do que a do sistema que obteve o segundo melhor resultado. Isto demonstra a eficácia e qualidade do sistema proposto. Outras vantagens do BroAD estão relacionadas com as informações providas e a abrangência da análise. Foram apresentadas as diferenças entre os resultados providos por cada um dos sistemas avaliados, mostrando que BroAD provê mais informações a respeito do comportamento do código analisado. Também foram mostradas as diferenças entre a abrangência de cada sistema, mostrando que BroAD pode analisar mais tipos de ataque.

6.1 Trabalhos Futuros

Diversas melhorias podem ser introduzidas para aprimorar a taxa de detecção do sistema BroAD, aumentar suas funcionalidades e torná-lo mais rápido para analisar uma quantidade maior de amostras. As principais são: i) a extensão da detecção por anomalia para tratar outras linguagens de programação ou tipos de aplicação; ii) a melhoria na detecção de *shellcodes*; iii) tornar o processo de análise mais rápido; iv) a identificação das vulnerabilidades que sofreram abuso e v) a geração de dados para mecanismos de proteção que fazem a detecção de código malicioso em tempo real.

Expandindo o componente de monitoramento de JavaScript para outras linguagens, principalmente VBScript, Java e ActionScript (linguagem de *script* usada dentro de arquivos do tipo Flash), será possível capturar o comportamento de amostras que utilizam estas linguagens e realizar a detecção por anomalia nestes casos.

Após a integração de outro componente que realiza captura do comportamento de códigos desenvolvidos em outras linguagens, a detecção de *shellcode* também poderia ser aplicada a estes comportamentos. Além disso, poderia ser utilizado outro emulador de instruções x86 em conjunto com a libemu, a fim de estender a capacidade desse tipo de detecção.

A otimização do processo de análise é importante para que o sistema seja capaz de analisar amostras mais rapidamente. Para alcançar isso é necessário melhorar o desempenho dos componentes quando executados conjuntamente.

A identificação das vulnerabilidades que sofreram abuso pelos códigos maliciosos auxilia no estudo destes e poderá ser feita com a união das técnicas propostas. Poderão ser feitas assinaturas que indiquem que o código manipulou certas funções de objetos ActiveX vulneráveis, por exemplo. Assim, caso a página seja considerada maliciosa, poderá se inferir que o ataque foi dirigido ao método desse objeto.

Outro trabalho futuro é a geração de dados para um sistema de detecção de código malicioso em tempo real. Este sistema seria responsável por bloquear ataques identificados

enquanto o usuário está utilizando seu navegador *Web* e BroAD seria responsável por gerar dados sobre os códigos maliciosos analisados visando auxiliar o sistema de proteção no processo de detecção.

6.2 Publicações

Durante a realização deste trabalho foram publicados três artigos científicos em uma conferência nacional e duas internacionais. Em [17] e [24], o foco é na obtenção de comportamentos maliciosos com o uso de uma ferramenta de análise de *malware* que utiliza o *driver* apresentado na Seção 4.1.2. Já em [1] são apresentados, de forma resumida, os principais resultados obtidos com BroAD.

Referências Bibliográficas

[1] V.M. Afonso, A.R.A. Grégio, D.S. Fernandes Filho, and P.L. de Geus. A hybrid system for analysis and detection of web-based client-side malicious code. In *Proceedings of the IADIS International Conference WWW/Internet 2011*, 2011.

[2] P. Baecher and M. Koetter. libemu-x86 shellcode detection and emulation. Disponível online em http://libemu.carnivore.it, 2011.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy*, pages 387–401. IEEE, 2008.

[4] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[5] M. Broesma. Web attacks slip under the radar. Disponível online em http://news.techworld.com/security/10620/web-attacks-slip-under-the-radar/, 2011.

[6] J. Butler and A. Anon. Bypassing 3rd party windows buffer overflow protection. *phrack Volume 0x0b, Issue 0x3e, Phile# 0x0, disponível online em http://phrack.com/issues.html?issue=62&id=5&mode=txt*, 7:13, 2004.

[7] S. Chenette. The ultimate deobfuscator. In *Proceedings of the ToorConX Conference, Seattle, WA, USA*, 2008.

[8] E. Chien. Malicious yahooligans. *Virus Bulletin, August*, 2006.

[9] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[10] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention

of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium-Volume 7*, pages 5–5. Usenix Association, 1998.

[11] R.S. Cox, S.D. Gribble, H.M. Levy, and J.G. Hansen. A safety-oriented platform for web applications. 2006.

[12] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, 2011.

[13] A. Dewald, T. Holz, and F.C. Freiling. ADSandbox: Sandboxing javascript to fight malicious websites. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1859–1864. ACM, 2010.

[14] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. *iNetSec 2009–Open Research Problems in Network Security*, pages 52–62, 2009.

[15] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106, 2009.

[16] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. USENIX Association, 2007.

[17] D.S. Fernandes Filho, A.R.A. Grégio, V.M. Afonso, R.D.C. Santos, M. Jino, and P.L. de Geus. Análise comportamental de código malicioso através da monitoração de chamadas de sistema e tráfego de rede. In *Anais do X Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 311–324, 2010.

[18] K. Fernandez and D. Pagkalos. Xssed.com. xss (cross-site scripting) information and vulnerable websites archive, 2007.

[19] Security Focus. Sina dloader class activex control 'donwloadandinstall' method arbitrary file download vulnerability. *Disponível online em http://www.securityfocus.com/bid/30223/*, 2008.

[20] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious flash advertisements. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 363–372. IEEE, 2009.

[21] M. Fossi, G. Egan, K. Haley, E. Johnson, T. Mack, T. Adams, J. Blackbird, M. Low, D. Mazurek, D. McKinney, and P. Wood. Symantec global internet security threat report. trends for 2010. *Volume XVI, April*, 2011.

[22] S. Frei, T. Duebendorfer, G. Ollmann, and M. May. Understanding the web browser threat. *TIK, ETH Zurich*, 2008.

[23] C. Grier, S. Tang, and S.T. King. Secure web browsing with the op web browser. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 402–416. Ieee, 2008.

[24] A.R.A. Grégio, D.S. Fernandes Filho, V.M., R.D.C. Santos, M. Jino, and P.L. de Geus. Behavioral analysis of malicious code through network traffic and system call monitoring. volume 8059, page 80590O. SPIE, 2011.

[25] M. Van Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2009.

[26] Z. Gyöngyi and H. Garcia-Molina. Web spam taxonomy. In *First International Workshop on Adversarial Information Retrieval on the Web*, 2005.

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[28] Ponemon Institute. Perceptions about network security. *Disponível online em http://www.juniper.net/us/en/local/pdf/additional-resources/ponemon-perceptions-network-security.pdf*, 2011.

[29] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 601–610. ACM, 2007.

[30] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[31] Z. Li, Y. Tang, Y. Cao, V. Rastogi, Y. Chen, and B. Liu. WebShield: Enabling Various Web Defense Techniques without Client Side Modifications. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[32] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: An attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 440–450. ACM, 2010.

[33] Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2. *Disponível online em http://support.microsoft.com/kb/875352*, 2008.

[34] J. Mieres. Phoenix explit's kit - from the mythology to a criminal business. Disponível online em http://www.malwareint.com/docs/pek-analysis-en.pdf, 2010.

[35] A. Moshchuk, T. Bragin, D. Deville, S.D. Gribble, and H.M. Levy. Spyproxy: Execution-based detection of malicious web content. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, page 3. USENIX Association, 2007.

[36] T. Müller, B. Mack, and M. Arziman. Wef - web exploit finder. *Disponível online em http://www.xnos.org/fileadmin/labs/wef/Whitepaper_WEF_Automatic_Drive_By_Download_Detection_English.pdf*, 2006.

[37] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[38] J. Nazario. Phoneyc: A virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, pages 6–6. USENIX Association, 2009.

[39] P.H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60. ACM, 2009.

[40] C. Reis, J. Dunagan, H.J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web (TWEB)*, 1(3):11–es, 2007.

[41] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.

[42] J.R. Rocaspana. Shelia: A client honeypot for client-side attack detection. *Disponível online em http://www.cs.vu.nl/˜herbertb/misc/shelia*, 2009.

[43] C. Seifert and R. Steenson. Capture - honeypot client (capture-hpc), 2006.

[44] H. Shacham, M. Page, B. Pfaff, E.J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[45] F. Valeur, G. Vigna, C. Kruegel, and E. Kirda. An anomaly-driven reverse proxy for web applications. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 361–368. ACM, 2006.

[46] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.

[47] J. Williams and D. Wichers. Owasp top 10–2010. *OWASP Foundation*, 2010.

[48] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

Glossary

Buffer overflow Tipo de vulnerabilidade de *software* em que pode-se escrever além dos limites de uma variável de forma a alterar dados de controle, normalmente o endereço de retorno de uma função.

Cookie Conjunto de dados passados em conexões HTTP, utilizados para controle de sessão.

Driver Programa que executa com altos privilégios, normalmente usado para fazer interface com dispositivos de *hardware*.

Exploit Conjunto de dados que abusa de alguma vulnerabilidade em um programa.

Heap Estrutura de dados que armazena objetos alocados na memória de um programa.

Hook Interceptação em um programa de forma a alterar seu fluxo de execução.

Link Uma referência a um endereço ou documento, local ou na Internet.

Malware Programas maliciosos.

NOP sled Sequência de instruções NOP (instrução que não realiza nenhuma operação) que servem apenas para levar o fluxo de execução até as instruções colocadas após esse conjunto.

Phishing E-mails *phishing* são utilizados para enganar os usuários, fazendo-os prover informações pessoais ou executar um determinado programa.

Plug-in de navegador *Web* Componente que estende as funcionalidades do navegador *Web*.

Safe browsing Serviço do Google que informa se uma dada URL contém códigos maliciosos.

Shellcode Trecho de código, normalmente escrito em linguagem de máquina, executado após o processo de abuso de um *software*. Seu nome vem do fato de nos primeiros casos ele ser usado para abrir um terminal de comando.

Site Página da *Internet*.

Smart screen filter Componente integrado ao Internet Explorer que verifica se as URLs acessadas pelo navegador fazem parte de uma lista de páginas maliciosas, impedindo o acesso em caso afirmativo.

Stack Estrutura de dados onde são alocados determinados objetos utilizados em um programa.

Stackguard Componente integrado ao compilador GCC que insere verificações na pilha de um programa a fim de verificar sua integridade.

Worm *Worm* é uma das classes de programas maliciosos e é caracterizada pela propagação de forma automática.