Detecção de ataques por ROP em tempo real assistida por *hardware*

Marcus Botacin[1], **Paulo Lício de Geus**[1], **André Grégio**[2]

[1]Instituto de Computação (IC)
Universidade Estadual de Campinas (Unicamp)
Campinas – SP – Brasil

[2]Departamento de Informática (DInf)
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

Abstract. *Code injection used to be one of the main attack vectors to subvert systems functioning. The adoption of non-executable pages supported by hardware (NX/XD) and data execution prevention (DEP) eliminated this problem in practice. However, attackers found another way of leveraging control flow deviation by chaining blocks of code (gadgets) through* `ret` *instructions. This is known as Return-Oriented Programming (ROP) and is currently the main injection vector. Recently, techniques based on Control Flow Integrity (CFI) and code length arose to counter such attacks with reasonable effectiveness. We hereby propose a Windows-based, hardware-assisted system, that overcomes limitations of state-of-the-art, hardware-assisted approaches to detect ROP attacks in realtime.*

Resumo. *A injeção de código foi um dos principais ataques contra sistemas computacionais. A adoção das páginas não-executáveis com apoio de hardware (NX/XD) e prevenção de execução de dados (DEP) eliminou o problema na prática. Entretanto, os atacantes passaram a desviar o fluxo de controle legítimo, encadeando blocos de código (gadgets) via instruções de retorno. Tal técnica, a Programação Orientada à Retorno (ROP), tornou-se o principal vetor de injeção. Recentemente, abordagens baseadas em integridade do fluxo de controle (CFI) e comprimento de gadgets surgiram para tratar tais ataques. Propomos aqui um sistema assistido por hardware, construído sobre sistemas Windows, que supera limitações de abordagens da literatura na detecção de ataques por ROP em tempo real.*

1. Introdução

A injeção de código figura como um dos maiores vetores de ataque contra sistemas computacionais, seja em sua vertente *Web* (p.ex., *SQL Injection* e *Cross Site Scripting*) ou de exploração da memória, como *buffer overflow/stack overflow*. Devido à popularidade dos ataques de injeção, vários mecanismos foram desenvolvidos de modo a se tentar bloqueá-los, tais como a inserção de canários na pilha, a desativação da execução de código em páginas de dados, pilha e heap (DEP, NX, XD) e a aleatorização de endereços-base (ASLR) de bibliotecas e módulos (que impede ataques com *offset* estático). Embora esses mecanismos tenham sido efetivos no bloqueio de execução de códigos externos, o abuso de sistemas ainda persiste, uma vez que os atacantes encontraram uma nova forma de subverter o fluxo de execução original através do reuso de código legítimo. Esta técnica,

conhecida como *Return-Oriented Programming* (ROP), consiste no encadeamento de trechos de código e permite todo tipo de computação.

Abordagens para se lidar com ROP se dividem entre: aplicáveis ao código-fonte através da instrumentação do compilador; *patching* de binários em tempo de execução; detecção em tempo real com auxílio de *hardware*. A primeira abordagem é de aplicação restrita, pois o código-fonte nem sempre está disponível; a segunda abordagem é extremamente dependente de arquitetura e pode sofrer de limitações para ser aplicada em *software* protegido e/ou com geração de código em tempo de execução; a terceira abordagem destaca-se por sua flexibilidade, embora sua implementação possa causar *overhead*. Neste trabalho, propõe-se implementações de técnicas para detecção de ataques por ROP em tempo real, no ambiente Windows[1], com auxílio de *hardware*, buscando minimizar o *overhead* de análise e desenvolvimento em comparação com as ferramentas estado-da-arte. As contribuições incluem, além da detecção dos ataques por ROP suportada por instruções do processador e da minimização de *overhead*, a introdução de uma técnica de monitoração do sistema como um todo, que não depende de informações *a priori* de aplicativos-alvo, tampouco de injetar código nos processos monitorados.

Este artigo é dividido da seguinte maneira: introduz-se na Seção 2 os conceitos básicos de ROP e os detalhes de funcionamento dos monitores de desempenho utilizados, bem como as abordagens existentes para a detecção de ataques por ROP e suas limitações; na Seção 3, discute-se o sistema proposto e os detalhes das técnicas utilizadas em sua implementação; na Seção 4, apresenta-se os testes realizados para validar a eficácia do sistema proposto, resultados obtidos e discussão, incluindo as limitações e outros ataques similares em potencial. Conclui-se o artigo na Seção 5.

2. Aspectos Técnicos e Trabalhos Relacionados

Nesta seção, são apresentados conceitos sobre Programação Orientada a Retorno (ROP) e monitoramento de desvios (*branches*), tecnologia usada na implementação da proposta deste artigo. São também apresentadas as abordagens já propostas na literatura para lidar com ataques ROP, classificadas de acordo com seu cenário de atuação (de acordo com [Bania 2010]). De modo geral, tais abordagens se propõem a reforçar alguma política de controle de fluxo (CFI) e se tornaram populares a ponto de serem implementadas em aplicativos, como Chromium/Google Chrome[2] e EMET [Microsoft 2013].

2.1. Conceitos

ROP. Ataques por injeção de código são uma das principais formas de desviar o fluxo normal de execução de aplicações para o propósito dos atacantes. As variações de *heap* e *stack overflow* mobilizaram a indústria em busca de soluções para barrar este tipo de ataque. As soluções propostas mitigaram o problema à medida em que se buscou detectar a injeção de código, através do uso de canários, impedir ataques baseados em *offset* fixos através de ASLR e impedir que os códigos injetados fossem executados (DEP, NX, XD). Contudo, os atacantes desenvolveram uma nova forma de desviar o fluxo de execução que não inclui executar diretamente código malicioso injetado, o que recoloca a atenção da comunidade sobre este tipo de ataque. Esses ataques baseiam-se em encadear uma sequência de instruções do código original de modo a se executar uma ação maliciosa. A execução não é impedida por nenhum mecanismo de proteção, uma vez que se

[1] Este sistema foi escolhido devido sua relevante fatia de mercado, afetando a maioria dos usuários. [2] `http://www.chromium.org/developers/testing/control-flow-integrity`

trata da execução de código legítimo, embora fora da especificação original. Tais ataques de reuso de código são conhecidos como ROP, dado que as sequências de instruções são usualmente terminadas em instruções de retorno (`ret`), as quais desviam o fluxo de execução para um outro ponto. As referidas sequências, denominadas *gadgets*, são geralmente formadas por poucas instruções que, quando combinadas, permitem a realização de uma computação completa (execução de uma ação). A Figura 1 exemplifica um ataque por ROP onde se subtrai dois valores da memória (`eax,edx`), resultando em `eax`.

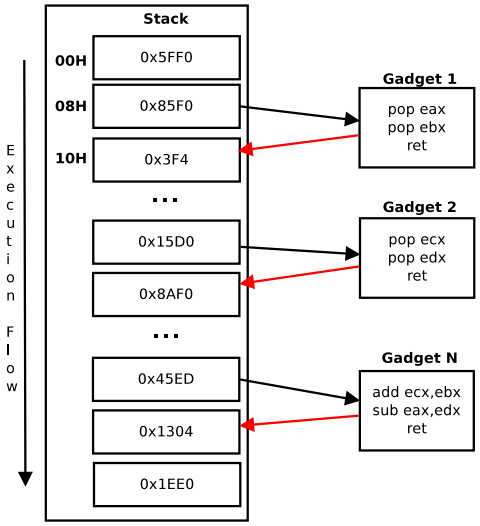

Figura 1. Ataque por ROP - subtração dos registradores EAX e EDX.

Monitoramento de *branch*. Processadores modernos de diferentes fabricantes e arquiteturas contam com monitores em *hardware* para obter informações de desempenho e funcionamento. Dentre eles, destaca-se o monitoramento de *branch*, capaz de armazenar informações como origem e destino de instruções de desvio de fluxo (p.ex., `JMP`, `CALL`, `RET`). Neste artigo, usa-se a tecnologia de monitoramento presente nos processadores Intel, por questões de disponibilidade. O monitor de *branch* é um recurso do processador que pode armazenar os últimos desvios executados em registradores específicos (MSR), em número limitado, ou em uma página do sistema operacional, limitado apenas por um *threshold* definido pelo desenvolvedor que, ao ser atingido, pode gerar uma interrupção. O monitoramento, por se tratar de recurso de *hardware*, introduz *overhead* teoricamente nulo. Além disso, conta com filtros que permitem selecionar o modo de captura (*kernel/userland*) e os tipo de ações (`CALL`, `RET`, `JMP`). O acesso aos dados é permitido apenas no nível do *kernel* e é em geral implementado por um *driver*. O uso dos recursos de monitoramento é independente de plataforma e, portanto, portável – requerindo apenas bibliotecas compatíveis para a interpretação dos dados, como nos procedimentos de *disassembly* e *dump* de memória.

2.2. Trabalhos Relacionados

A seguir, são mostradas algumas abordagens para lidar com ataques por ROP.

Abordagens em tempo de compilação/Reescrita de código. Para mitigar a exploração causada em um programa por um ataque por ROP, as abordagens em tempo de compilação visam, em geral, impedir a construção de determinados tipos de *gadgets*, que estes estejam acessíveis ou que o compilador crie certas estruturas de controle mais vulneráveis. [Bletsch et al. 2011a] apresentam *control flow locking* (CFL), uma técnica

alternativa de proteção que cria um marcador para as estruturas de transição. Seu funcionamento é semelhante ao de um *mutex*, que deve ser ativado sempre que o código precisar ser executado. A idéia é que códigos oriundos de ataques por ROP se insiram em um ponto sem passar por este marcador, causando a detecção da execução do código ilegítimo. Sua implementação é por reescrita do código *assembly* gerado. *Gfree* [Onarlioglu et al. 2010], por sua vez, trata do problema das instruções desalinhadas, as quais podem ser convertidas em *gadgets* para ataques por ROP. O sistema implementa a substituição dessas instruções usando um pré-processador para o *GNU Assembler*. As abordagens citadas são de aplicação restrita, pois dependem do código-fonte e deixam sistemas legados desprotegidos, por não serem compatíveis com programas já compilados.

Instrumentação. Para contornar algumas limitações e ser aplicável a *software* pré-existente, muitas abordagens se baseiam na instrumentação do binário para a detecção de ataques por ROP. [Davi et al. 2009] propõe uma arquitetura de verificação dinâmica (DynIMA) que modifica o conteúdo do programa em tempo de carregamento para incluir instruções de *tainting* para monitoração. *DROP* [Chen et al. 2009] baseia-se em interceptar a execução de instruções `ret` e verificar o suposto *gadget*. Sua implementação é alcançada por instrumentação no *Valgrind*[3]. *Ropguard* [Fratric 2012] não visa detectar os *gadgets* em si, mas a transição destes para APIs que possam vir a ser usadas para propósitos maliciosos, como a criação de um processo no sistema. A monitoração é realizada através da injeção de DLL. *ROPMEMU* [Graziano et al. 2016] implementa um emulador capaz de reconstruir os fluxos de execução e analisar ataques ROP a partir de *dumps* de memória física. Esta solução é altamente dependente e limitada pela implementação dos mecanismos de emulação e aquisição de memória. Tais abordagens, embora aplicáveis a qualquer binário, sofrem tanto com as já estudadas limitações de seus mecanismos de monitoração quanto com o fato do *overhead* ser proibitivo em alguns cenários de uso.

Reescrita de binário. A tentativa de superar as limitações dos mecanismos de instrumentação resultou na técnica de reescrita de binários já existentes, a qual pode ser aplicada uma vez durante seu primeiro uso de forma a gerar um binário instrumentado. ILR [Hiser et al. 2012] expande a ideia de aleatorização de endereços de dados (ASLR) para instruções, de modo que ataques com *offsets* codificados estaticamente não funcionem. Estratégia semelhante é proposta em [Pappas et al. 2012] e [Wartell et al. 2012]. Já [Zhang et al. 2013] propõem uma abordagem que consiste em uma política na qual desvios indiretos são proibidos, exceto para uma série de posições especificadas por uma *white-list*. Contudo, as técnicas baseadas em reescrita de binários ainda apresentam limitações quando aplicadas a binários que geram códigos dinamicamente, como os que executam em interpretadores ou máquinas virtuais.

Monitoramento com suporte de *hardware*. Uma abordagem de funcionamento mais amplo é o uso de recursos de *hardware* para a construção de um mecanismo de monitoramento, pois não dependem do código-fonte ou da reescrita do binário e atuam em tempo real. *Hypercrop* [Jiang et al. 2011] é uma máquina virtual com suporte de *hardware* (HVM) que pode ser utilizada para monitorar o sistema de forma ampla e identificar um ataque ROP. No entanto, dado seu grande número de interrupções, o *overhead* de monitoramento se torna proibitivo. Abordagens com baixo *overhead*, de maneira geral, têm se baseado no uso de contadores de desempenho, em especial o monitor de *branch* mencionado anteriormente. [Yuan et al. 2011] e [Kompalli 2014] se utilizam dos dados

[3] `http://valgrind.org/`

de desempenho providos pelo `perf`[4] do Linux e *VTune Amplifier*[5] da Intel, respectivamente, para a identificação de ataques. Embora exibam exemplos bem sucedidos, os mecanismos usados não são totalmente apropriados para o tipo de monitoramento pretendido. Isto causou o surgimento de ferramentas de propósito específico, como *ROPecker* e *KBouncer*. *ROPecker* [Cheng et al. 2014] é um módulo para o *kernel* do Linux capaz de detectar ataques por ROP durante sua execução. Sua lógica consiste de uma janela deslizante que verifica o comprimento da cadeia de *gadgets* e atua em duas etapas: a criação de um banco de dados de *gadgets* por meio do *disassembly* da aplicação e a verificação dos acessos usando o recurso de *last branch record* (LBR) dos processadores. *KBouncer* [Pappas et al. 2013] detecta ataques por ROP usando LBR para identificar se uma chamada de função veio de um *gadget* ou não. Para tanto, as chamadas de função dos aplicativos monitorados são interceptadas por meio da biblioteca *Detours*[6]. Os aplicativos a serem monitorados podem ser escolhidos através de uma interface semelhante à do Microsoft EMET[7]. As políticas de detecção se baseiam no comprimento da cadeia e na existência de instruções `RET` sem a respectiva `CALL`. Embora as abordagens no estado-da-arte tenham avançado os métodos de detecção de ataques por ROP, estas ainda apresentam sérias limitações, sobretudo na aplicação do mecanismo LBR (provê informação limitada) e pela injeção de código no binário monitorado ser uma escolha de projeto. A proposta do presente artigo é a de tratar adequadamente tais limitações.

3. Arquitetura e Implementação

Para alcançar os objetivos estabelecidos para este trabalho, propõe-se um sistema de monitoramento baseado em arquitetura cliente-servidor, onde o servidor é um *driver* de *kernel* responsável por obter as informações do sistema e o cliente é um programa responsável por interpretar as informações obtidas.

O *driver* captura as informações diretamente do *hardware*, utilizando o monitor de *branch*. O monitor utilizado é o *Branch Trace Store* (BTS), em vez do LBR usado em outras abordagens da literatura, pois permite definir o *threshold* de armazenamento e a geração de uma interrupção. Assim, definiu-se este para uma instrução de desvio, que interrompe o sistema sempre que esta ocorre, além de permitir a identificação do processo que a causou. Os dados capturados são enfileirados em ordem.

O cliente se comunica com o *driver* via IRP, retirando os dados da fila criada pelo lado servidor. O processo de introspecção é realizado para cada tupla (endereços de origem e destino da instrução de desvio) de modo a identificar a biblioteca ou processo base de um endereço-alvo. Isto é feito por meio de chamada à `GetModuleHandle`—para lidar com ASLR—e pelas ferramentas *Dumpbin*[8] e/ou *DLL Export Viewer*[9]—para conhecer a função chamada por meio de seu *offset*.

É importante ressaltar que a abordagem proposta não requer injeção de código em nenhum processo. Além disso, dado que a captura de dados é feita de modo *system-wide*, todos os processos podem ser monitorados simultaneamente. Por fim, a adoção de BTS no lugar de LBR elimina o risco da sobrescrita de valores. Mais detalhes sobre o sistema proposto, em sua versão inicial e estendida, são descritos a seguir.

[4] `https://perf.wiki.kernel.org`
[5] `https://software.intel.com/en-us/intel-vtune-amplifier-xe`
[6] `http://research.microsoft.com/en-us/projects/detours/`
[7] `https://support.microsoft.com/en-us/kb/2458544`
[8] `https://support.microsoft.com/en-us/kb/177429`
[9] `http://www.nirsoft.net/utils/dll_export_viewer.html`

3.1. Sistema Inicial

Conforme mencionado, o cliente é responsável por obter os dados capturados pelo servidor e processá-los de acordo com regras estabelecidas. O cliente pode filtrar os dados por processo ou atuar de maneira abrangente, monitorando todas as aplicações em execução. Neste caso, embora se tenha uma área de proteção maior, o desempenho do sistema piora devido ao maior processamento requerido. Para minimizar este problema, força-se a execução do cliente em um núcleo do processador diferente do núcleo sob monitoração, não gerando penalidades de desempenho para a aplicação monitorada. O cliente pode optar por implementar diferentes políticas, como as discutidas a seguir. Quando o cliente identifica uma violação de política, emite um alerta através de seu *daemon* (Figura 2).

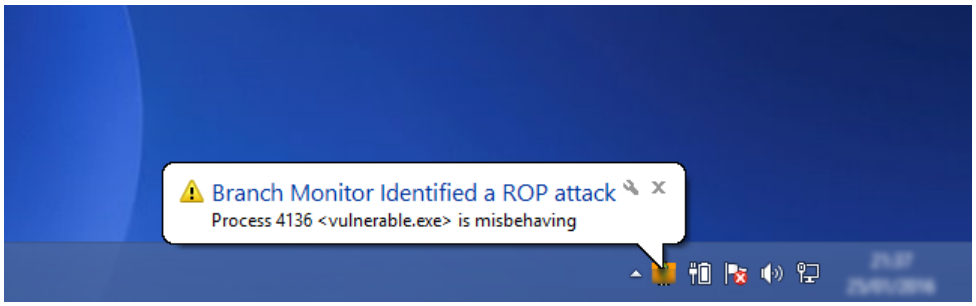

Figura 2. Notificação de ataque identificado emitida pelo cliente de monitoração.

3.1.1. A Política do "CALL−RET"

Esta política pode ser implementada para processos sob monitoração por meio da decodificação do primeiro *byte* da instrução apontado pelo endereço capturado pelo monitor de *branch*, identificando seu tipo. Para efeitos de comparação e facilidade de implementação, considerou-se para esta proposta a mesma versão da política usada por *KBouncer* [Pappas et al. 2013]. Essa versão é considerada menos restritiva, pois desconsidera casos degenerados, por exemplo, o uso de instruções `JMP` e `POP+RET` para fazer o papel de chamadas do tipo `CALL`. A Tabela 1 reproduz os *opcodes* relativos às instruções do tipo `CALL`[10], enquanto que a Tabela 2 reproduz os da instrução `RET`[11].

Tabela 1. *Opcodes* de instruções do tipo `CALL`.

Opcode	E8 cw	E8 cd	FF /2	9A cd	9A cp	FF /3
Mnemonic	CALL rel16	CALL rel32	CALL r/m16 ou r/m32	CALL ptr16:16	CALL ptr16:32	CALL m16:16 ou m16:32

Tabela 2. *Opcodes* de instruções do tipo `RET`.

Opcode	C3	CB	C2 iw	CA iw
Mnemonic	RET	RET	RET imm16	RET imm16

A Listagem 1 ilustra correspondências de instruções `RET` com `CALL` anteriores, o que configura um fluxo de execução íntegro. Em caso de desvio de fluxo por ataque por ROP, seriam vistas instruções `RET` sem as respectivas `CALL`. Os endereços (`FROM` e `TO`) são obtidos diretamente através do monitor BTS. O *framework* proposto é responsável por obter o PID, através da interrupção, os *bytes* `INSTR`, através da leitura da memória correspondente aos endereços, e interpretá-los (mnemônicos `CALL` e `RET`), através das Tabelas 1 e 2.

[10] `http://x86.renejeschke.de/html/file_module_x86_id_26.html` [11] `https://pdos.csail.mit.edu/6.828/2004/readings/i386/RET.htm`

Listagem 1. Correspondências "CALL-RET" indicando fluxo de execução íntegro.

```
1 PID 3140 FROM   6b8e7f17 INSTR e8 - CALL
2 PID 3140 TO     6b9d90c1 INSTR c3 - RET
3 PID 4196 FROM   77b2ce8e INSTR e8 - CALL
4 PID 4196 TO     77aa591e INSTR c2 - RET
5 PID 2532 FROM   3de50b6c INSTR e8 - CALL
6 PID 2532 TO     40714979 INSTR c2 - RET
```

3.1.2. A Política do Comprimento do *gadget*

Apesar de efetiva em muitos casos, a política de `CALL-RET` não é suficiente para cobrir todos os casos de ataques por ROP. Logo, abordagens adicionais fazem-se necessárias. Implementou-se então no sistema a política de comprimento de *gadget*, também disponível no Kbouncer. Esta política consiste em identificar uma sequência de *gadgets* curtos, característica dos *payloads* de ataque por ROP, dado que sequências longas tendem a incorrer em muitas mudanças de estado, dificultando seu sequenciamento para propósitos maliciosos. Na abordagem proposta, foi implementada uma janela de 20 instruções (mesmo tamanho usado por KBouncer).

A Listagem 2 ilustra o tamanho dos blocos de código de programas legítimos, obtidos manualmente a partir da memória (lida com `ReadProcessMemory`[12] e interpretados pela *LibOpcodes*[13] e/ou *Capstone*[14]). A Listagem 3 reproduz o tamanho dos *gadgets* do *exploit* ROP tal qual utilizado pelo Kbouncer. Nota-se que, para programas legítimos, o tamanho dos blocos[15] é significativamente superior ao *threshold* estabelecido, diferentemente dos *gadgets* ROP.

Listagem 2. Comprimento dos blocos em programas legítimos (em número de instruções).

```
1 PID 3820 FROM 5f0dea04 TO 5f0deb30 INSTR 15
2 PID 3820 FROM 5f0deb4a TO 5f0deb53 INSTR 21
3 PID 3820 FROM 5f0dea0e TO 5f0dea17 INSTR 19
```

Listagem 3. Tamanho dos *gadgets* encontrados em programa atacado por ROP (em número de instruções).

```
1 ADDR 7C3411C0 INSTR 2
2 ADDR 7C3415A2 INSTR 1
3 ADDR 7C34252C INSTR 2
4 ADDR 7C346C08 INSTR 1
5 ADDR 7C378C84 INSTR 3
```

3.1.3. Discussão Preliminar

Embora implemente as mesmas políticas presentes no *KBouncer*, o sistema proposto neste artigo possui diferenças que o tornam sensivelmente mais amplo no âmbito de proteção das aplicações. Por exemplo, enquanto o *KBouncer* destina-se à proteção de aplicações de forma independente, o sistema proposto neste é capaz de monitorar mais de um processo

[12] `https://msdn.microsoft.com/en-us/library/windows/desktop/ms680553(v=vs.85).aspx` [13] `https://github.com/Groundworkstech/pybfd` [14] `http://www.capstone-engine.org` [15] em número de instruções

simultaneamente (modo *system-wide*). Essa característica faz com que não seja necessário injetar código no processo monitorado, ao contrário do *KBouncer*, que requer a injeção de uma DLL no processo monitorado para a redireção por *Detours*.

Além disso, o sistema proposto não depende da chamada de APIs para a avaliação de registros de *branch*, a qual ocorre quando há interrupção pelo mecanismo BTS. *KBouncer*, por sua vez, exige a redireção do fluxo para uma chamada de API previamente interceptada (via *hooking*). Outra diferença relevante do sistema proposto é a capacidade de armazenar registros de *branch* ilimitados devido ao mecanismo de BTS. Já o *KBouncer* é limitado aos registradores do mecanismo LBR (atualmente 16 registradores na arquitetura Haswell). [Carlini and Wagner 2014] apontam esta limitação do *KBouncer* como uma possível forma de se evitar a detecção de ataques por ROP.

Apesar de geralmente efetivas, as políticas disponíveis no *KBouncer* e no sistema aqui proposto não são capazes de lidar com todos os casos de ataques ROP, ou estão suscetíveis a técnicas de evasão. [Göktaş et al. 2014] discute que, apesar de elevar o nível de dificuldade para a efetividade dos ataques, abordagens baseadas no comprimento dos *gadgets* são de difícil calibração—um comprimento muito grande pode permitir que ataques sejam bem sucedidos e comprimentos muito pequenos podem impedir aplicações legítimas de funcionar. O autor discute em detalhes tanto o conceito de comprimento do *gadget* como das cadeias que estes formam. [Carlini and Wagner 2014] mostram como é possível construir longas sequências de *gadgets*, tornando assim os *payloads* de ROP mais parecidos com blocos de código de aplicações legítimas. Portanto, requer-se esforços complementares para se lidar de forma consistente com ataques do tipo ROP.

3.2. Sistema Estendido

Como alternativa para lidar com ataques por ROP, destaca-se o surgimento de heurísticas com base em determinar a frequência global de desvios de fluxo encontrados em aplicações em operação normal e sob ataque. Dentre as abordagens disponíveis, considerou-se a apresentada por [Ferreira et al. 2014], que instrumenta aplicações por meio do *framework* PIN [Luk et al. 2005] para determinar e verificar o limiar da referida frequência. O sistema estendido conta com implementação auxiliada por *hardware* e baseia-se no sistema de monitoração de *branch* já apresentado para detectar ataques por ROP. A comparação entre a abordagem de Ferreira et al. (com PIN) e o sistema estendido deste artigo explicitam as vantagens do último: não-dependência de injeção de código (feita pelo PIN); uso em múltiplas aplicações sem a necesssidade de instrumentação individual; *overhead* de captura teoricamente nulo, devido ao mecanismo de monitoração de *branch* em *hardware*. Na prática, todo *overhead* adicionado deve-se ao processamento da política. Para não impactar no desempenho da aplicação monitorada, pode-se executar o monitor em um núcleo de CPU diferente do da aplicação, como feito em [Quinn 2012].

3.2.1. Implementação

Na implementação do sistema estendido, levou-se em conta as considerações apresentadas por [Ferreira et al. 2014] como base de funcionamento do algoritmo de janela. O funcionamento do sistema aqui proposto é análogo à implementação com PIN, ou seja, baseia-se na interpretação dos blocos de dados resultantes da interrupção gerada pelo mecanismo de BTS. Os dados fornecidos pelo BTS são os endereços de origem e de destino do desvio. Assim, considerando duas interrupções consecutivas, tem-se o ponto de entrada e o de saída em um bloco de código, o qual pode ser entendido como os blocos

Listagem 4. Identificação de um bloco de código através da diferença entre dois endereços consecutivos. Neste caso, o bloco tem início em `0x48ff5ab8` e término em `0x48ff5ac0`.

```
1 PID: 4876 FROM: 48ff5ab0 TO: 48ff5ab8
2 PID: 4876 FROM: 48ff5ac0 TO: 48ff5ad0
```

básicos (*Basic Blocks* - BBLs) do PIN. A leitura do conteúdo de memória no bloco permite a obtenção das instruções intermediárias, pois sabe-se que a extremidade do bloco é uma instrução de desvio. Tem-se então um desvio por bloco e, de acordo com o tamanho da janela a ser aplicada ao bloco (ou a múltiplos blocos), pode-se contar a frequência de desvios como no algoritmo proposto originalmente.

A Listagem 4 ilustra a identificação de blocos a partir de *branches* consecutivos. Na linha 1, inicia-se a execução do bloco residente no endereço `0x48ff5ab8`. Na linha 2, na interrupção seguinte, tem-se que o fluxo sai do bloco pela instrução presente no endereço `0x48ff5ac0`. Desta forma, sabemos que foram executados 8 *bytes* (`c0-b8`) de instruções. Deve-se realizar o *dump* de memória da instrução correspondente a esses *bytes* para identificar as instruções executadas.

A Figura 3 ilustra quatro blocos de código distintos, apresentando, portanto, quatro instruções de desvio. Para janelas de 16 instruções, por exemplo, tem-se que a frequência de desvios pode variar de um a três em 16.

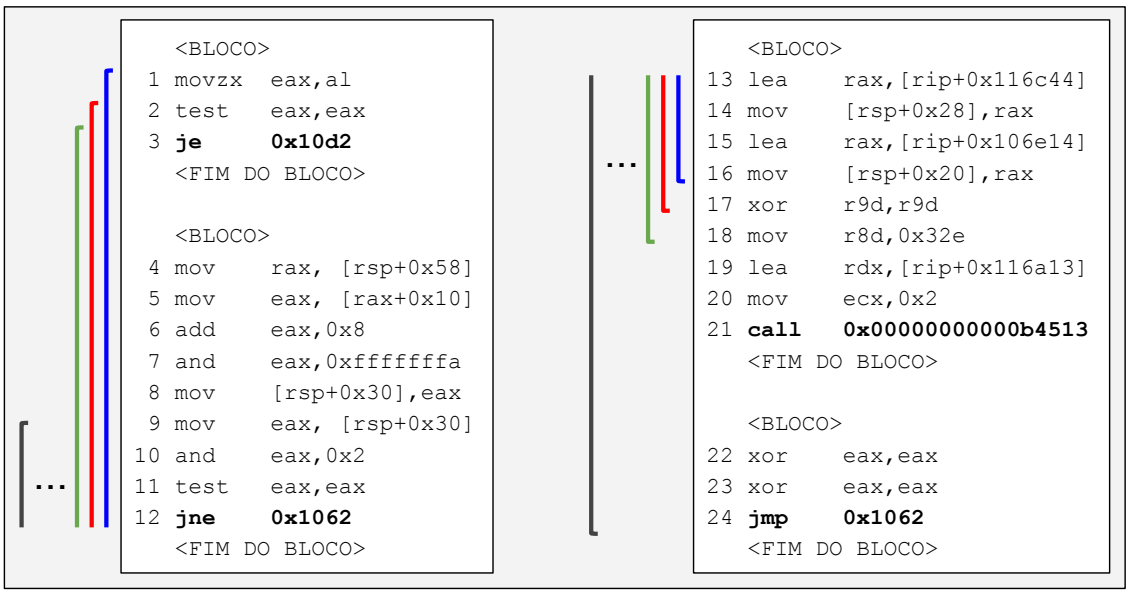

Figura 3. Blocos com instruções de desvios nas extremidades (em negrito) e janela deslizante de 16 instruções.

4. Testes e Resultados

Para validar a aplicação da solução proposta para detecção de ataques por ROP com casos reais, usou-se o *exploit* de Nguyen [Son 2011] em ambiente de testes contendo a vulnerabilidade descrita pelo CVE 2011-0065 [CVE 2011]. O referido *exploit* faz uso de ROP para se evadir das técnicas de proteção do Windows, permitindo a exploração de uma falha de *use-after-free* no navegador Firefox 3.5. A escolha deste *exploit* deu-se pela

disponibilidade de uma amostra cuja execução funcionasse em sistemas operacionais modernos (Windows 7 e 8), ainda que de 32 bits (necessário para o sucesso do ataque de *heap spray*). Cabe enfatizar que a predominância de amostras de ataque por ROP tem por alvo outros sistemas operacionais (Windows XP e Linux), impedindo a comparação direta do sistema proposto com outras abordagens existentes.

Nos testes, amparados pelas discussões e estudos de caso da literatura, foram aplicadas as configurações ótimas definidas pelos respectivos autores. A execução do *exploit* disponível resultou em *crash* do navegador e na ativação do sistema proposto causada pelas métricas de comprimento de *gadget* e densidade de desvios. Os resultados são detalhados a seguir.

4.1. Comprimento de *gadget*

A Tabela 3 mostra um trecho da janela de instruções de desvio capturada pelo mecanismo de monitoração de *branch* durante a execução do *exploit*. Observa-se a execução de pequenos *gadgets* de dois bytes cada (entradas `0x7c346c0a-0x7c346c0b` e `0x7c37a140-0x7c37a141`.

Tabela 3. Janela de instruções de desvio contendo 2 *gadgets* de 2 *bytes* e 2 instruções.

FROM	TO
—-	0x7c346c0a
0x7c346c0b	0x7c37a140
0x7c37a141	—-

A fim de prover uma análise qualitativa do fato observado, fez-se o *disassembly* estático da biblioteca *MSVCR71.dll 7.10.3052.4 - 32bits* com a ferramenta *objdump*, resultando na Listagem 5.

Listagem 5. Código legítimo (alinhado) contendo uma sequência de bytes que pode ser abusada em um ataque *(gadget).*

```
1 7c346c08:     f2 0f 58 c3             addsd   %xmm3,%xmm0
2 7c346c0c:     66 0f 13 44 24 04       movlpd %xmm0,0x4(%esp)
```

Considerando o salto para o endereço `0x7c346c0a`, desalinhado, obtém-se o *gadget* real executado (*bytes* \x58\xc3), como mostrado na Listagem 6.

Listagem 6. Código desalinhado contendo o *gadget* realmente executado.

```
1 0x1000 (size=1)   pop     rax
2 0x1001 (size=1)   ret
```

4.2. Densidade de desvios

Para a verificação da densidade de desvios, usou-se uma janela de 32 instruções, uma vez que esta é a que provê maior poder de identificação [Ferreira et al. 2014]. Neste teste, cujos resultados são apresentados na Figura 4, observa-se que aplicações limitadas por *I/O*, tais como “adobe”, “soffice” e “mplayer”, apresentam baixa densidade de desvios. Uma aplicação limitada pela *CPU*, como a “superpi”, apresenta densidade de desvios superior, ainda que dentro do *threshold* estabelecido (10 instruções de desvio). Aplicações como o navegador Internet Explorer (“iexplo”) apresentam limitações tanto de *I/O*, ao se conectar ao servidor remoto, quanto de *CPU*, ao renderizar a página, e por isso apresentam taxa

de desvios mediana. O *exploit* apresentou a maior taxa de desvios dentre os calculados (superior ao limiar), conforme esperado.

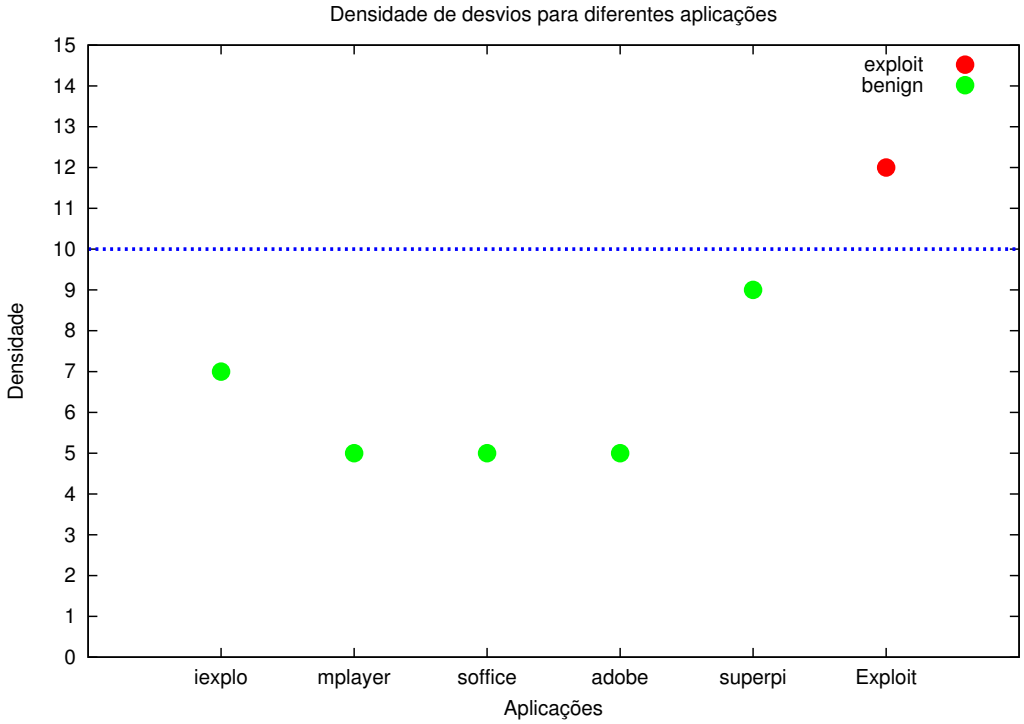

Figura 4. Densidade de desvios em aplicações "benignas" e no *exploit* de Nguyen.

4.3. Discussão

Devido à natureza dinâmica dos ataques ROP, as dificuldades em sua mitigação são inúmeras. O fato do código executado ser "legítimo" impacta na detecção do ataque e na criação de heurísticas, pois pode impedir que aplicações não maliciosas funcionem corretamente. A monitoração do fluxo de execução é dificultada por haver várias formas de se realizá-la, com saltos diretos e indiretos. Bloquear tais desvios, contudo, é impraticável—muitas formas de código polimórfico, compilação *Just-In-Time* (JIT), entre outros, se baseiam nesse tipo de construção.

Abordagens baseadas em tempo de compilação não são suficientes para o bloqueio deste tipo de ameaça, uma vez que novas formas de se construir os *gadgets* são frequentemente encontradas [Schuster et al. 2014]. Ademais, novas formas de ataques são propostas periodicamente, como os ataques de programação orientada a saltos (JOP) [Bletsch et al. 2011b] e a laços (LOP) [Lan et al. 2015]. Portanto, uma solução para os ataques por ROP deve ser encarada com abordagem multi-fatorial, como as empregadas para ataques por *buffer overflow*. Nestas, a solução passa por abordagens em tempo de compilação (canários), em nível de sistema operacional (ASLR) e em nível de *hardware* (DEP/NX e, mais recentemente, MPX [Intel 2013]).

Dado o cenário apresentado dos ataques por ROP, o sistema proposto neste artigo é capaz de complementar as demais abordagens existentes. Além disso, o monitoramento de *branch* pode ser feito mais facilmente com base no cumprimento de hipóteses acerca das garantias providas por outros fatores de proteção (externos ou do sistema operacional).

Considerando as limitações na detecção de ataques por ROP, a monitoração pelo mecanismo de *branch* impõe vários desafios, alguns deles não completamente solucionados. Um exemplo é o tratamento de transição para o espaço de *kernel*, onde há perda de controle do endereçamento de retorno devido ao mecanismo de BTS ter sido desabilitado no *kernel*. A perda do endereço de retorno, entretanto, não afeta o detector de ataques por ROP, pois apenas os blocos extremos (de transição) são afetados e estes são significativamente menores do que todas as janelas utilizadas. Processo semelhante ocorre quando se usa comunicação entre processos (IPC) de forma direta, mas este não é o cenário de aplicação de *exploits* baseados em ROP. É possível implementar a habilitação do mecanismo para detecção de ataques no nível do *kernel*, porém isso requer esforço adicional na interpretação dos blocos, já que os dados precisam ser filtrados e ameaças neste nível estão fora do escopo deste trabalho. Seu tratamento é deixado como trabalho futuro.

Finalmente, existem outros ataques que envolvem a computação baseada em *gadgets*. Construções do tipo *weird computing* [Vanegue 2014, Bangert et al. 2013, Shapiro et al. 2013] podem vir a ser a próxima forma de exploração, dado que possuem características similares aos ataques por ROP (como o uso de elementos legítimos).

4.4. Desempenho

O *overhead* adicionado pela solução é dependente da aplicação e do número de instâncias monitoradas. Em um caso de monitoração onde todo o *disassembly* é realizado em tempo de execução, este pode atingir valores de até 43%. Tais valores, contudo, são inferiores às soluções estado-da-arte, que atingem valores de até 100%. Maiores informações sobre o desempenho da solução proposta podem ser encontradas na página *web* da solução[16].

5. Conclusão

Neste artigo, introduziu-se uma nova solução de monitoramento em tempo real de ataques por ROP, a qual não requer injeção de código em processos ou recompilação/instrumentação de binários. A solução proposta foi utilizada para implementar algoritmos de detecção por integridade de fluxo de controle e heurística de detecção por comprimento de *gadgets*. Implementou-se também uma abordagem heurística para o cálculo da frequência de instruções de desvios, que mostrou-se suficiente para diferenciar execuções de blocos de códigos benignos de um *exploit* baseado em ROP. Fazer uso de suporte de *hardware*, como no caso do sistema proposto, é um passo importante para o aprimoramento das técnicas de detecção deste tipo de ataque.

Agradecimentos

Os autores agradecem o apoio recebido do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq via Projeto MCTI/CNPq/Universal-A 14/2014 (Processo 444487/2014-0) e da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES, em especial via Projeto FORTE - Forense Digital Tempestiva e Eficiente (Processo: 23038.007604/2014-69 - Edital 24/2014 - Programa Ciências Forenses).

Referências

Bangert, J., Bratus, S., Shapiro, R., and Smith, S. W. (2013). The page-fault weird machine: Lessons in instruction-less computation. In *Proc. of the 7th USENIX Conf. on Offensive Technologies*, WOOT'13, pages 13–13.

[16] `https://sites.google.com/site/branchmonitoringproject/`

Bania, P. (2010). Security mitigations for return-oriented programming attacks. https://www.kryptoslogic.com/download/ROP_Whitepaper.pdf. Acesso em junho/2016.

Bletsch, T., Jiang, X., and Freeh, V. (2011a). Mitigating code-reuse attacks with control-flow locking. In *Proc. of the 27th Annual Computer Security Applications Conf.*, ACSAC '11, pages 353–362, New York, NY, USA. ACM.

Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. (2011b). Jump-oriented programming: A new class of code-reuse attack. In *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40.

Carlini, N. and Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In *Proc. of the 23rd USENIX Conf. on Security Symposium*, SEC'14, pages 385–399.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). Drop: Detecting return-oriented programming malicious code. In *Proc. of the 5th International Conf. on Information Systems Security*, ICISS '09, pages 163–177, Berlin, Heidelberg. Springer-Verlag.

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., DENG, H., et al. (2014). Ropecker: A generic and practical approach for defending against rop attack. *Network and Distributed System Security Symposium*.

CVE (2011). Cve-2011-0065. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0065. Acessado em junho/2016.

Davi, L., Sadeghi, A.-R., and Winandy, M. (2009). Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proc. of the 2009 ACM Workshop on Scalable Trusted Computing*, STC '09, pages 49–54, New York, NY, USA. ACM.

Ferreira, M. T., Filho, A. S., and Feitosa, E. (2014). Controlando a frequência de desvios indiretos para bloquear ataques rop. *Anais do SBSEG 2014*.

Fratric, I. (2012). Runtime prevention of return-oriented programming attacks. https://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf. Acessado em junho/2016.

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA. USENIX Association.

Graziano, M., Balzarotti, D., and Zidouemba, A. (2016). ROPMEMU: A framework for the analysis of complex code-reuse attacks. In *ASIACCS 2016, 11th ACM Asia Conference on Computer and Communications Security, May 30-June 3, 2016, Xi'ian, China*.

Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. (2012). Ilr: Where'd my gadgets go? In *Proc. of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, Washington, DC, USA. IEEE Computer Society.

Intel (2013). Introduction to intel® memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions. Acessado em junho/2016.

Jiang, J., Jia, X., Feng, D., Zhang, S., and Liu, P. (2011). Hypercrop: A hypervisor-based countermeasure for return oriented programming. In *Proc. of the 13th International Conf. on Information and Communications Security*, ICICS'11, pages 360–373, Berlin, Heidelberg. Springer-Verlag.

Kompalli, S. (2014). Using existing hardware services for malware detection. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 204–208.

Lan, B., Li, Y., Sun, H., Su, C., Liu, Y., and Zeng, Q. (2015). Loop-oriented programming: A new code reuse attack to bypass modern defenses. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 190–197.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In

Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '05, pages 190–200, New York, NY, USA. ACM.

Microsoft (2013). Introducing Enhanced Mitigation Experience Toolkit (EMET) 4.1. `http://blogs.technet.com/b/srd/archive/2013/11/12/introducing-enhanced-mitigation-experience-toolkit-emet-4-1.aspx`. Acessado em junho/2016.

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. (2010). G-free: Defeating return-oriented programming through gadget-less binaries. In *Proc. of the 26th Annual Computer Security Applications Conf.*, ACSAC '10, pages 49–58, New York, NY, USA. ACM.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2012). Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *Proc. of the 22Nd USENIX Conf. on Security*, SEC'13, pages 447–462, Berkeley, CA, USA. USENIX Association.

Quinn, R. (2012). *Detection of malware via side channel information*. PhD thesis, Binghamton University.

Schuster, F., Tendyck, T., Pewny, J., Maaß, A., Steegmanns, M., Contag, M., and Holz, T. (2014). Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium*, RAID 2014, pages 88–108.

Shapiro, R., Bratus, S., and Smith, S. W. (2013). "weird machines"in elf: A spotlight on the underappreciated metadata. In *Proc. of the 7th USENIX Conf. on Offensive Technologies*, WOOT'13, pages 11–11.

Son, N. H. (2011). Rop chain for windows 8. `http://security.bkav.com/home/-/blogs/rop-chain-for-windows-8/normal`. Acessado em junho/2016.

Vanegue, J. (2014). The weird machines in proof-carrying code. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 209–213.

Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. (2012). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of the 2012 ACM Conf. on Computer and Communications Security*, CCS '12, pages 157–168.

Yuan, L., Xing, W., Chen, H., and Zang, B. (2011). Security breaches as pmu deviation: Detecting and identifying security attacks using performance counters. In *Proc. of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 6:1–6:5, New York, NY, USA. ACM.

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. (2013). Practical control flow integrity and randomization for binary executables. In *Proc. of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA. IEEE Computer Society.

