Universidade Federal do Amazonas
Instituto de Computação
Programa de Pós-Graduação em Informática

Ailton da Silva dos Santos Filho

Reduzindo a Superfície de Ataque dos Frameworks de Instrumentação Binária Dinâmica

Manaus
Março de 2019

Ailton da Silva dos Santos Filho

Reduzindo a Superfície de Ataque dos Frameworks de Instrumentação Binária Dinâmica

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas como requisito para obtenção do grau de Mestre em Informática.

Orientador: Prof. Dr. Eduardo Luzeiro Feitosa

Manaus
2019

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Santos Filho, Ailton da Silva

S237r Reduzindo a Superfície de Ataque dos Frameworks de Instrumentação Binária Dinâmica / Ailton da Silva Santos Filho. 2018

96 f.: il. color; 31 cm.

Orientador: Eduardo Luzeiro Feitosa

Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Instrumentação Binária Dinâmica. 2. Anti-Instrumentação. 3. Malware. 4. Malware consciente de análise. I. Feitosa, Eduardo Luzeiro II. Universidade Federal do Amazonas III. Título

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FOLHA DE APROVAÇÃO

"Reduzindo a Superfície de Ataques dos Frameworks de Instrumentação Binária Dinâmica"

AILTON DA SILVA DOS SANTOS FILHO

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Eduardo Luzeiro Feitosa - PRESIDENTE

Prof. Eduardo James Pereira Souto - MEMBRO INTERNO

Prof. André Ricardo Abed Grégio - MEMBRO EXTERNO

Manaus, 11 de Março de 2019

Agradecimentos

Agradeço a todos os meus professores, especialmente a meu orientador pelo constante suporte e orientação, a Universidade Federal do Amazonas e seus colaboradores, técnicos, à seção administrativa, a FAPEAM que disponibilizou recursos para minhas pesquisas, a Universidade de Zaragoza e professor Ricardo Rodríguez por terem me recebido tão bem e fornecido inspiração, aos meus amigos, e à minha família pelo apoio incondicional e paciência.

Resumo

Aplicações maliciosas, comumente chamadas de *malware*, posam como um dos problemas mais relevantes no cenário tecnológico atual, sendo consideradas a raiz de muitas ameaças de segurança na Internet. Em parte, isso se deve a capacidade dos desenvolvedores de *malwares* de prontamente responder ao surgimento de novas soluções de segurança, desenvolvendo artefatos para evitá-las. Nesta dissertação, são apresentadas contramedidas para mitigar recentes mecanismos, chamados de técnicas de anti-instrumentação, empregados para evitar à análise de *malwares.* Dentre elas, esta dissertação foca nas técnicas que exploram o aumento da superfície de ataque produzido pelas ferramentas de instrumentação binária dinâmica, permitindo ataques como execução de código arbitrário *(Arbitrary Code Execution*), o que se converte em fuga da instrumentação binária dinâmica (DBI), no contexto de ferramentas DBI. Para garantir a eficácia das contramedidas propostas, provas de conceito foram desenvolvidas e testadas em ambiente controlado com um conjunto de técnicas anti-instrumentação. Por fim, foi feita uma análise no impacto no desempenho inserido pelo uso das abordagens propostas. Como resultado, esta dissertação demonstra que é possível reduzir a superfície de ataque explorável das ferramentas DBI através da mitigação das técnicas anti-instrumentação. No entanto, isso não garante necessariamente a transparência de tais ferramentas.

Palavras-chave: Anti-instrumentação, Malware consciente de análise, Malware, Instrumentação Binária Dinâmica, Anti-análise

Abstract

Malicious applications pose as one of the most relevant issues in today's technology scenario, being considered the root of many Internet security threats. In part, this owes the ability of malware developers to promptly respond to the emergence of new security solutions by developing artifacts to avoid them.

In this work, we present countermeasures to mitigate one of the recent mechanisms used by malware to avoid detection of anti-instrumentation techniques. Among these techniques, this work focuses on those that increase the attack surface of malicious applications, allowing attacks such as arbitrary code execution, which becomes an escape of dynamic binary instrumentation (DBI) in the context of DBI tools. To ensure the effectiveness of the proposed countermeasures, proofs of concept were developed and tested in a controlled environment with a set of anti-instrumentation techniques. Finally, an analysis was made on the performance impact of using the proposed approaches. As a result, this dissertation demonstrates that it is possible to reduce the exploitable attack surface of DBI tools by mitigating anti-instrumentation techniques. However, this does not necessarily guarantee the transparency of such tools.

Keywords: Anti-instrumentation, Analysis-aware, Malware, Dynamic Binary Instrumentation, Anti-analysis

Lista de Figuras

2.1 Arquitetura do framework DBI Pin. 9
2.2 Estrutura da memória de um processo. 11
2.3 Estrutura da memória de um processo modificada para representar a presença de DBI. 12
2.4 Ilustração do conteúdo de um Cache de Código em relação ao código original. 14

3.1 Classificação resultante para as técnicas de evasão 22
3.2 Árvore de processos para uma aplicação sob análise do Pin. 26
3.3 Varredura em memória por assinaturas de Cache de Código. 27
3.4 Listagem das seções de um processo sob análise do Pin. 30
3.5 Comparação entre a estrutura TLS com e sem o Pin. 32

5.1 Diagrama de classes do `PinVMShield`. 49
5.2 Ilustração da TLS sendo utilizada por duas threads de um processo. . . 50

Lista de Tabelas

3.1 Relação de Técnicas de Evasão por Trabalho 37

3.2 Continuação da Relação de Técnicas de Evasão por Trabalho 38

4.1 Relação de Técnicas de Evasão com Contramedidas 46

5.1 *Overhead* introduzido pelo `PinVMShield` com as contramedidas propostas. 57

Sumário

1 Introdução **2**

1.1 Motivação 3

1.2 Objetivos 4

1.3 Contribuições 5

1.3.1 Organização da Dissertação 5

2 Fundamentação Teórica **6**

2.1 Instrumentação Binária Dinâmica 6

2.2 Frameworks DBI 7

2.2.1 Intel Pin 8

2.2.2 DynamoRIO 10

2.3 Exploração dos Frameworks DBI 11

2.3.1 Cache de Código 14

2.4 Malwares Evasivos 16

2.5 Histórico da Evasão dos *Frameworks* DBI 17

3 Técnicas de Evasão **19**

3.1 Classificação das Técnicas Anti-Instrumentação 19

3.2 Técnicas Diretas de Evasão de Frameworks DBI 22

3.2.1 Artefatos de Cache de Código 22

3.2.1.1 Detecção de EIP/RIP 23
3.2.1.2 Código Automodificante 23
3.2.1.3 Contexto Inesperado 24
3.2.1.4 Exceções Inesperadas 24
3.2.1.5 *Xmode Code* 24
3.2.1.6 Abuso das Instruções de *syscall* 25
3.2.1.7 Emulação incorreta da instrução *rdfsbase* 25
3.2.1.8 Negligenciamento do No-eXecute Bit (nx) 25
3.2.2 Artefatos de Ambiente 25
3.2.2.1 Detecção do Processo Pai 26
3.2.2.2 *Memory Fingerprinting* / Detecção de Cache de Código 26
3.2.2.3 Detecção Por *Argv* 27
3.2.2.4 *Handle* de Processo 28
3.2.2.5 Número de *File Handlers* 28
3.2.2.6 Número Máximo de *File Handlers* Abertos 28
3.2.2.7 Pico no Uso de Memória 28
3.2.2.8 Alteração no Número Máximo de *File Handlers* 29
3.2.2.9 Detecção pela Máscara de Sinal (*Signal Masks*) 29
3.2.2.10 *Fingerprinting pinvm.dll*/ Detecção de Bibliotecas do DynamoRIO 29
3.2.2.11 Detecção Pelas Funções Exportadas Por *Pintools* 29
3.2.2.12 Detecção Pelos Nomes de Seções 30
3.2.2.13 Detecção Por *Event Handles* 30
3.2.2.14 Detecção Por *Section Handles* 31
3.2.2.15 Detecção por *Thread Local Storage* (TLS) 31
3.2.2.16 Variáveis de Ambiente Internas do Pin 31
3.2.3 Detecção do Compilador JIT 32

3.2.3.1 *DLL Hooks* . . . 32
3.2.3.2 Detecção Pelas Permissões de Páginas de Memória . . . 33
3.2.3.3 Detecção por Chamadas Comuns a *APIs* . . . 33
3.2.4 Detecção de *Overhead* . . . 34
3.3 Técnicas Indiretas de Evasão . . . 34
3.3.1 Limitação de Recursos . . . 34
3.3.1.1 *Stalling Code* . . . 35
3.3.2 Limitação de Funcionalidade . . . 35
3.3.3 Instruções Não Suportadas . . . 35
3.3.4 *Crash Code*/Comportamentos Não Suportados . . . 36
3.4 Discussão . . . 36

4 Mitigação a Evasão de Frameworks DBI 40
4.1 Detecção de EIP/RIP . . . 40
4.2 Código Automodificante . . . 41
4.3 Detecção do Processo Pai . . . 41
4.4 *Memory Fingerprinting*/Detecção de Cache de Código . . . 42
4.5 Detecção pelas Funções Exportadas por *Pintools* e Detecção pelos Nomes de Seções . . . 42
4.6 *DLL Hooks* . . . 43
4.7 Detecção pelas Permissões de Páginas de Memória . . . 43
4.8 Detecção por Chamadas Comuns a APIs . . . 44
4.9 Detecção de *Overhead* . . . 44
4.10 Stalling Code . . . 45
4.11 Discussão . . . 45

5 Novas Contramedidas 48
5.1 *PinVMShield* . . . 48

5.2 Contramedidas 49
5.2.1 Detecção por *Thread Local Storage* (TLS) 50
5.2.1.1 *Thread Local Storage* (TLS) 50
5.2.1.2 *Contramedida* 51
5.2.2 *Memory Fingerprinting* / Detecção de Cache de Código 52
5.2.2.1 Cache de Código 52
5.2.2.2 Contramedida 52
5.2.3 Negligenciamento do *No-eXecute Bit* (nx) 53
5.2.3.1 *No-eXecute Bit* (nx) 54
5.2.3.2 Contramedida 54
5.3 Eficácia e Desempenho da Contramedidas Propostas 55
5.4 Discussão 57

6 Considerações Finais **59**
6.1 Conclusões 60
6.2 Dificuldades Encontradas 60
6.3 Tópicos em Aberto 61

Referências Bibliográficas **63**

A Provas de Conceito das Técnicas Anti-Instrumentação **68**
A.1 Negligenciamento do No-eXecute Bit (nx) 68
A.2 Memory Fingerpriting / Detecção de Cache de Código 70
A.3 Detecção por Thread Local Storage (TLS) 72

B Provas de Conceito das Contramedidas **75**
B.1 Negligenciamento do No-eXecute Bit (nx) 75
B.2 Memory Fingerpriting / Detecção de Cache de Código 77
B.3 Detecção por Thread Local Storage (TLS) 81

Capítulo 1

Introdução

As aplicações desenvolvidas especialmente com propósitos maliciosas (comumente referenciadas como *malwares*) têm aumentado em quantidade e em complexidade durante os últimos anos (AV-TEST GmbH, 2018). Este fato supõe um problema para as empresas de antivírus, que precisam dispor de uma base de dados atualizada de *malwares* conhecidos para prover proteção eficiente a seus clientes. Dessa forma, analistas de segurança se deparam diariamente com um número crescentes de amostras a serem analisadas. Por exemplo, a Kaspersky reportou que em 2017 foram analisadas 360.000 aplicações maliciosas por dia (Kaspersky Lab, 2017).

Embora os *malwares* apresentem evoluções tanto em quantidade quanto em sofisticação, seu processo de análise é realizado manualmente ou automaticamente. A análise manual requer inicialmente uma análise estática (sem executar o código amostra, onde se analisa artefatos estáticos e reescreve-se o código binário da aplicação (Nethercote, 2004)) e em seguida uma análise dinâmica, onde são observados o comportamento do programa, sua interação com o sistema operacional e com a rede, durante sua execução. No entanto, a abordagem manual tende a ser complexa e a consumir tempo substancialmente dos analistas de *malwares*, tornando-se inapropriada para lidar com a crescente quantidade de aplicações maliciosas. Assim, as análises automatizadas em ambientes isolados de análise, como máquinas virtuais e emuladores, vêm se tornando cada vez mais populares (Greamo and Ghosh, 2011), por permitirem a análise de grandes quantidades de amostras sem intervenção humana.

Ambos os procedimentos de análise, manual e automático, podem adotar diferentes técnicas estáticas e dinâmicas de análise. Dentre as técnicas de análise dinâmica, a Instrumentação Binária Dinâmica (*Dynamic Binary Instrumentation*), ou simplesmente DBI, vem sendo empregada em diversas soluções de segurança, como em *taint analysis* (Stamatogiannakis et al., 2015), *unpacker* de *malwares* (Mariani et al., 2016), proteção da transparência de máquinas virtuais (Gilboy, 2016; Rodriguez et al., 2016), análise de *malwares* (Kulakov, 2017), dentre outras aplicações.

A Instrumentação Binária Dinâmica permite a análise em tempo de execução de aplicações, a nível binário, através da inserção de instruções arbitrárias (de análise) adicionais no código da aplicação, permitindo que o analista de segurança altere o comportamento da aplicação sob análise, inspecione estruturas como registradores e a pilha do processo, em tempo de execução, dentre outras capacidades. Sua crescente adoção deve-se, em parte, às vantagens que oferece: (i) ser independente de linguagem de programação e compilador utilizados para a geração da aplicação cliente - programa a ser analisado ; (ii) não requerer acesso ao código fonte da aplicação a ser analisada (Nethercote, 2004); (iii) exercer controle absoluto sobre a aplicação cliente e código executado (Rodriguez et al., 2016).

No entanto, à medida que as ferramentas baseadas em DBI ganharam popularidade, os desenvolvedores de *malwares* começaram a incorporar artefatos em seus códigos especialmente construídos para detectar ambientes de análise e ferramentas DBI. Dessa forma, para evitar ser detectado, o *malware* esconde seu comportamento malicioso quando em análise, se passando por uma aplicação benigna. Recentemente, Polino et al. (2017) realizaram a avaliação de 7.006 amostras de *malwares*, usando análise automatizada, onde foi observado que 15.6% das aplicações utilizavam ao menos uma técnica com o objetivo de evadir análises realizadas por ferramentas DBI. Os *malwares* que empregam tais artifícios são conhecidos como *malwares* evasivos (*evasive malware*), *malwares* cientes de análise (*analysis-aware malware*) ou *malwares* com personalidade dividida (*split personality malware*) (Balzarotti et al., 2010; Vishnani et al., 2011; Kumar et al., 2012; Rodriguez et al., 2016; Polino et al., 2017).

1.1 Motivação

Agentes e grupos desenvolvedores de *malwares* possuem interesse em evitar sua detecção, especialmente porque quanto mais tempo um *malware* permanecer indetectável, maior será a receita obtida e os danos causados.

Como resposta à popularização das técnicas anti-instrumentação, os analistas de segurança vêm intensificando seus esforços para desenvolver mecanismos que tornem suas ferramentas de análise mais transparentes em relação às aplicações sendo analisadas. Kirat et al. (2011) e Kirat et al. (2014) propuseram executar as amostras primeiramente em *"hardwares reais"* (*bare-metal*), ou seja, sem uso de máquinas virtuais, e posteriormente comparar o comportamento da amostra com quando é executada em um ambiente virtualizado.

Outros autores propuseram ferramentas, `PinVMShield` (Rodriguez et al., 2016) e `Arancino` (Polino et al., 2017), que identificam e impedem tentativas de evasão e detecção de ambientes de análise de *malwares*. Ambas ferramentas são baseadas em *instrumentação binária dinâmica*, o que permite aos analistas de segurança inserir códigos arbitrários durante a execução da aplicação sendo analisada. Embora poderosa, este tipo de técnica de análise é detectável pelos *malwares* através de uma variedade de artefatos deixados pelas ferramentas DBI no sistema operacional, como reportado na literatura. A seção 2.5 revisa os trabalhos que introduziram tais técnicas de evasão,

apresentando os mecanismos envolvidos.

Recentemente, Zhechev (2018) e Sun et al. (2016) passaram a questionar se ferramentas baseadas em DBI são apropriadas para a análise de *malwares*. Esses autores afirmam que através de técnicas de evasão anti-DBI específicas, é possível não só detectar as ferramentas que empregam instrumentação binária dinâmica, mas também explorar a *superfície de ataque*[1] inserida pelo uso dessas ferramentas, reduzindo a segurança dos ambientes que as utilizam. Segundo eles, a redução dos níveis de segurança é atribuída a possibilidade que *bugs* de software que seriam naturalmente difíceis de serem explorados no mundo real por um *malware*, se tornem vulnerabilidades que permitem a execução de códigos arbitrários (*Arbitrary Code Execution Vulnerabilities*) (Anley et al., 2011). No âmbito das ferramentas de instrumentação binária dinâmica, tanto a exploração das estruturas dos *frameworks* DBI como vulnerabilidades, quanto a execução de códigos arbitrários, se enquadram como fuga da instrumentação (Sun et al., 2016; Zhechev, 2018).

É nesse contexto que este trabalho se insere, investigando o comprometimento dos ambientes de análise que utilizam instrumentação binária dinâmica, realizável por técnicas de evasão anti-instrumentação específicas, incorporadas por aplicações maliciosas.

1.2 Objetivos

O objetivo desta dissertação é desenvolver métodos de proteção contra a fuga da instrumentação binária dinâmica (DBI), através do isolamento ativo, em tempo de execução e lógico entre as ferramentas DBI e as aplicações instrumentadas, a fim de permitir a integridade da análise de aplicações evasivas que utilizam técnicas anti-instrumentação.

Para tanto, é utilizada a solução `PinVMShield`, uma ferramenta com arquitetura de *plugins*, projetada para mitigar *malwares* evasivos. A eficácia dos métodos propostos é demonstrada através de testes realizados com provas de conceito das técnicas de evasão presentes na literatura e os impactos no desempenho computacional inseridos por seu uso são avaliados utilizando a ferramenta *SPEC CPU 2006* - largamente utilizada na literatura para medir o *overhead* computacional de ferramentas DBI.

Especificamente, pretende-se:

- Desenvolver soluções para isolar, logicamente, as aplicações sob análise da ferramenta DBI das estruturas alvejadas pelas técnicas de evasão que exploram estruturas de dados ou demais artefatos que podem levar a fuga da instrumentação ou ao comprometimento do ambiente de análise DBI;

- Atestar que a integridade das ferramentas DBI é mantida, se protegidas suas estruturas críticas quando contra as técnicas anti-instrumentação existentes.

[1] Manadhata (2008) definiu superfície de ataque como sendo o subconjunto de recursos do sistema que um código malicioso pode utilizar para atacar um sistema, ou seja artefatos relacionados às vulnerabilidades de um sistema

1.3 Contribuições

Esta dissertação alcançou as seguintes contribuições:

- Avaliação das técnicas de evasão anti-DBI propostas na literatura, resultando em uma revisão em formato de *survey*, a fim de identificar as técnicas que podem levar ao comprometimento das ferramentas de instrumentação binária dinâmica e a fuga da instrumentação;
- Desenvolvimento de provas de conceito (PoC) das técnicas de evasão, apresentadas apenas teoricamente na literatura, que podem levar à fuga da instrumentação binária dinâmica e não possuem contramedidas, com o objetivo de serem utilizadas na validação das soluções propostas;
- Desenvolvimento de um protótipo de proteção contra técnicas de evasão que exploram a superfície de ataque dos *frameworks* DBI, especificamente no ambientes Windows através do *framework* DBI Pin;
- Demonstração da capacidade do *framework* DBI Pin de manter seu isolamento e integridade quando garantida a proteção das suas estruturas críticas.

1.3.1 Organização da Dissertação

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta parte da teoria relacionada a instrumentação binária dinâmica, suas aplicações, principais *frameworks* existentes, como se dá a evasão das ferramentas DBI e a fuga da instrumentação, concluindo com um histórico dos trabalhos que apresentaram as técnicas anti-instrumentação.

O Capítulo 3 detalha as técnicas anti-instrumentação apresentadas na literatura, iniciando pelas classificações para tais técnicas, seguindo descrevendo em detalhes os princípios de funcionamento de cada técnica anti-instrumentação e apresentando as técnicas evasivas com capacidade de exploração das vulnerabilidades dos *frameworks* de instrumentação binária dinâmica.

O Capítulo 4 relata as contribuições de diversos trabalhos na mitigação das técnicas anti-instrumentação, através do desenvolvimento de contramedidas. O Capítulo 5 apresenta as soluções propostas como contramedida, descreve a implementação de provas de conceito e conclui com uma discussão sobre a eficácia das soluções propostas e sobre como elas afetam a superfície de ataque inserida pelas ferramentas DBI.

O Capítulo 6 conclui a dissertação com uma discussão sobre o uso dos *frameworks* DBI para propósitos de segurança de sistemas, as dificuldades encontradas ao longo do desenvolvimento deste trabalho e os tópicos em aberto.

Capítulo 2

Fundamentação Teórica

Este Capítulo apresenta a base teórica relacionada a *malwares* evasivos, instrumentação binária dinâmica (DBI) e ferramentas (*frameworks*) DBI, a fim de permitir a compreensão das técnicas de evasão, seus mecanismos de funcionamento e a fuga da instrumentação binária dinâmica.

A seção 2.1 descreve a técnica de análise Instrumentação Binária Dinâmica, ou *Dynamic Binary Instrumentation* (DBI), de acordo com a tese de doutorado de Nethercote (Nethercote, 2004) e com a definição de outros autores, mostrando os princípios de funcionamento e discutindo as mudanças que ocorrem no *layout* do espaço de memória de um processo sob instrumentação em relação a sua execução nativa. Em seguida, a seção 2.2 apresenta os dois *frameworks* DBI alvejados pelas técnicas de evasão presentes na literatura: Pin Luk et al. (2005b) e DynamoRIO Bruening et al. (2001). A seção 2.4 apresenta as definições dos termos *analysis-aware malware*, *split personality malware* e *evasive malware*, familiarizando o leitor com os termos.

2.1 Instrumentação Binária Dinâmica

Para definir Instrumentação Binária Dinâmica, ou *Dynamic Binary Instrumentation* (DBI), Nethercote (2004) quebrou a expressão, definindo-a termo a termo. *Dynamic analysis* envolve analisar uma aplicação cliente - aplicação sob análise - enquanto é executada, através de instrumentação. O termo instrumentação refere-se ao ato de inserir códigos adicionais a um programa. *Binary analysis* refere-se a analisar um programa no nível de código de máquina, representado como código binário ou código executável (*object code ou executable code*). Assim, *Dynamic Binary Instrumentation* pode ser definido como uma técnica de análise que ocorre durante a execução do programa cliente, onde artefatos de código, conhecidos como *analysis code*, são inseridos na aplicação sendo analisada por um processo externo.

De acordo com Nethercote (2004), a Instrumentação Binária Dinâmica possui como vantagens, frente a outras técnicas de análise: (i) não requer qualquer preparação prévia a análise no programa cliente - programa a ser instrumentado -, como injeção de

bibliotecas ou alterações no código binário da aplicação cliente; (ii) cobre naturalmente com a análise todo o código da aplicação cliente, o que pode não ser possível nas análises estáticas, especialmente quando a aplicação gera códigos dinamicamente; (iii) por ser binária, não requer acesso ao código fonte da aplicação cliente ou recompilação da mesma. Rodriguez et al. (2016) acrescentam como vantagens, ainda: ser independente de linguagem de programação e compilador utilizados para a geração da aplicação cliente; possuir controle absoluto sobre a execução da aplicação a ser analisada.

Por essas vantagens, a instrumentação binária dinâmica passou a ser largamente empregada, especialmente para a segurança de sistemas, em aplicações como *taint analysis* (Stamatogiannakis et al., 2015), *unpacker* de *malwares* (Mariani et al., 2016), proteção da transparência de máquinas virtuais (Rodriguez et al., 2016) e análise de *malwares* (Kulakov, 2017).

De acordo com Nethercote (2004), no entanto, a Instrumentação Binária Dinâmica possui duas principais desvantagens. A primeira é o custo computacional da instrumentação que incorre em tempo de execução, o que pode afetar a execução das aplicações sendo analisadas. A segunda é ser de difícil implementação, uma vez que é uma tarefa árdua para o desenvolvedor de reescrever um código executável em tempo de execução. Zhechev (2018) acrescenta ainda que as ferramentas DBI podem não se apropriadas para análises de códigos potencialmente maliciosos, ou *malwares*, uma vez que não conseguem garantir, em seu modo atual, propriedades de segurança importantes, como a transparência e o isolamento, em relação a aplicação sob análise.

Para reduzir esses impactos negativos, os *frameworks* DBI fornecem plataformas bem documentadas e um padrão de desenvolvimento de ferramentas DBI. A seção seguinte aborda os dois principais *frameworks* DBI da atualidade e os principais alvos de técnicas anti-instrumentação na literatura.

2.2 Frameworks DBI

Os *frameworks* DBI começaram a surgir no início dos anos 2000 (Bruening et al., 2001; Luk et al., 2005b; Nethercote and Seward, 2003) com o objetivo de fornecer um conjunto extensível de ferramentas de instrumentação para abstrair complexidades de implementação e melhorar o desempenho das ferramentas DBI. Esta subseção apresenta os dois *frameworks* DBI que são alvejados pelas técnicas de evasão presentes na literatura: Pin (Luk et al., 2005b) e DynamoRIO (Bruening et al., 2001). Dessa forma, outras ferramentas DBI, como Valgrind (Nethercote, 2004), que não foram alvejadas por técnicas anti-DBI não serão tratadas no momento.

É interessante notar que Rodriguez et al. (2016) e Zhechev (2018) justificam o enfoque das técnicas anti-instrumentação nessas soluções devido o grande número de ferramentas de análise desenvolvidos para esses *frameworks*, além de serem as soluções que impõe o menor custo computacional adicional sobre a aplicação sendo analisada.

2.2.1 Intel Pin

O *framework* Intel Pin (Luk et al., 2005b) é uma solução que permite o desenvolvimento de ferramentas de instrumentação binária dinâmica portáveis entre sistemas operacionais e hardwares, transparentes e computacionalmente eficientes. Foi desenvolvido pela Intel em 2005 com suporte aos sistemas operacionais Windows, Linux e MacOS X, para as arquiteturas IA-32, x86-64 e MIC. É um dos mais populares *frameworks* de instrumentação binária dinâmica por causar a menor sobrecarga computacional sobre aplicações que fazem uso intensivo da CPU (Rodríguez et al., 2014), contando com mais de 300.000 downloads e mais de 700 citações (Intel Corporation, 2013).

O Pin possui um conjunto de APIs (*Application Programming Interface*) que torna possível escrever ferramentas de instrumentação (*pintool*) carregadas no mesmo espaço de endereço da aplicação alvo, permitindo assim a análise do estado da aplicação em, por exemplo, conteúdo dos registradores utilizados, memória e fluxo de execução.

Além de análise, o Pin fornece ferramentas para alterar o programa alvo durante a execução, sobrescrevendo instruções e o conteúdo de registradores e de valores na memória. Cada *pintool* é composta por um componente, conhecido como rotina de instrumentação, que decide onde e qual código deve ser inserido e pelo código que será inserido, conhecido como rotina de análise. As APIs do Pin permitem que sejam utilizados quatro níveis diferentes de granularidade de análise que indicam a menor unidade sob análise do executável. São elas:

1. **Instrução**: O modo instrumentação de instruções permite que sejam executadas análises instrução a instrução, com o lado negativo de incrementar o *overhead* da solução.

2. ***Trace***: Permite que sejam inseridas instruções de análise em *traces*, trechos de código que iniciam com um destino de desvio (*branch*) tomado e terminam com um desvio incondicional, podendo ter múltiplas saídas (Intel Corporation, 2017c).

3. **Imagem**: Permite a inserção de instruções de análise do instrumentador em imagens. Imagens representam todas as estruturas de dados correspondentes a um executável. Bibliotecas compartilhadas (*Shared libraries*) também são representadas como imagens (Intel Corporation, 2017a).

4. **Rotina**: O nível de granularidade por rotina permite que sejam inseridas instruções de analisem em rotinas, ou seja, em representações de funções, rotinas e procedimentos, tipicamente produzidas por compiladores para linguagens de programação procedurais (Intel Corporation, 2017b).

É possível executar o Pin em dois modos diferentes: em tempo de execução e em modo de sonda. Em modo de instrumentação dinâmica em tempo de execução, o Pin nunca executa o código original da aplicação a ser instrumentada, sendo carregado em memória apenas para referência, mas sim o código gerado pelo compilador *just-in-time* (JIT). Assim, é possível uma análise de alta granularidade, instrução a instrução. Esse

código reside no espaço de memória conhecido como Cache de Código (*Code Cache*) e contém as instruções originais da aplicação mais modificações determinadas pelo Pin e pela *pintool*. Em modo de sonda, são inseridas instruções de salto para as rotinas de análise no código original do cliente carregado em memória.

O Pin é composto por três componentes típicos de ferramentas DBI, como mostrado na Figura 2.1: (1) a aplicação a ser instrumentada, em amarelo; (2) a ferramenta DBI desenvolvida com o Pin, normalmente chamada de *pintool*, em amarelo; e (3) o motor (engine) do *framework* DBI, de cor azul.

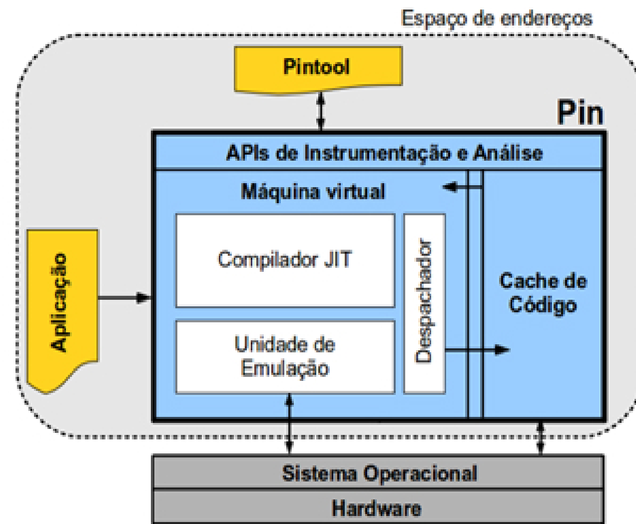

Figura 2.1: Arquitetura do framework DBI Pin (Luk et al., 2005b).

O motor DBI consiste de uma máquina virtual (*virtual machine* ou VM), uma cache de código (*code cache*) e uma interface de programação de aplicativos (API) utilizada pelas *pintools*. A VM toma como entrada o código executável nativo do aplicativo a ser instrumentado e usa um compilador *just-in-time* (JIT) para inserir o código de instrumentação, imediatamente antes da execução. Então, o código instrumentado resultante é salvo, através do despachador, em uma da caches de código e o fluxo de execução do programa é transferido para ele. Após a execução desse trecho de código, o compilador JIT busca a próxima sequência de instruções a ser executada e gera mais código. A unidade de emulação é encarregada de tratar as instruções que não podem ser executadas diretamente, como as chamadas de sistema, que requerem um tratamento especial da máquina virtual. Além disso, a Figura 2.1 ilustra a forma como o Pin se relaciona com o sistema operacional, a aplicação instrumentada e a *pintool*, além de explicitar que a aplicação cliente, a pintool e o motor DBI são executados no mesmo espaço de endereços virtuais, o que é a raiz de diversos problemas de transparência dos frameworks DBI, segundo Zhechev (2018).

A entrada passada para o Pin é o próprio executável alvo, ou seja, a aplicação alvo da análise tem sua execução iniciada pelo Pin. Porém, é possível realizar instrumentação

sobre uma aplicação já carregada em memória e em execução. O *framework* intercepta a execução da primeira instrução da aplicação instrumentada e toma o controle do fluxo de execução, assim a execução da aplicação - incluindo chamadas ao sistema e interrupções - é compassada pelo Pin e pelas operações realizadas pela *pintool*.

Além das vantagens da instrumentação binária dinâmica, Luk et al. (2005b) destaca que o Pin é o *framework* de instrumentação que apresenta o menor *overhead* - custo computacional adicional de processamento - frente a outros *frameworks*, como Valgrind (Nethercote and Seward, 2003) e DynamoRIO (Bruening et al., 2001), o que permitiu que o Pin fosse empregado largamente utilizado pela comunidade científica e em soluções comerciais de análise de aplicações (por exemplo, a ferramenta Intel ©Inspector que é um *debugger* de memória e *threads* para as linguagens C, C++ e Fortran). O Pin é parte integrante dos conjuntos de ferramentas Intel ©Parallel Studio XE e Intel ©System Studio.

2.2.2 DynamoRIO

DynamoRIO (Bruening et al., 2001) é um *framework* de instrumentação/otimização binária dinâmica derivado do *Dynamo*, uma ferramenta de otimização dinâmica, desenvolvido para os sistemas operacionais Windows, Linux, Android e arquiteturas IA-32/AMD64/ARM/AArch64. É distribuído sem custos sob a licença BSD.

Da mesma forma que o Pin, DynamoRIO não é limitado a fazer análises em tempo de execução, sendo capaz de alterar o programa alvo durante a execução, sobrescrevendo instruções, o conteúdo de registradores e de valores na memória. Porém, não possui diferentes modos de execução, como o Pin. DynamoRIO executa de uma forma intermediária aos dois modos do Pin. Assim, sempre é utilizada a cache de código e o compilador *just-in-time*. No entanto, o código presente nessa cache é idêntico ao original. Apenas códigos de controle de fluxo de execução são alterados, além dos explicitamente modificados pela ferramenta de instrumentação. DynamoRIO tem como entrada o binário a ser instrumentado. A partir daí, o código é copiado para a cache de código conforme demandado pelo fluxo de execução. Da mesma forma que o Pin, qualquer interação da aplicação cliente com o sistema operacional é controlada pelo instrumentador.

DynamoRIO possui como principais vantagens, a otimização nativa de *hot traces*, ou seja, trechos de código executados muitas vezes e em que não há ou praticamente não há modificações no código entre execuções e realizar poucas alterações no código original da aplicação cliente, o que o torna ideal para o desenvolvimento de ferramentas que realizam instrumentação de baixa granularidade (*lightweight*) (Nethercote, 2004).

A fim de garantir sua transparência e evitar técnicas anti-instrumentaçãp, o DynamoRIO adota um conjunto de princípios de transparência ao realizar a instrumentação: (i) realizar o menor número de modificações possível no código da aplicação cliente; (ii) esconder as modificações realizadas; (iii) isolar recursos do *framework* DBI. No entanto, como descrito na seção 3, esses princípios adotados não são suficientes para garantir a transparência do *framework* DBI, uma vez que existe uma variedade de técnicas anti-instrumentação eficazes contra o DynamoRIO.

2.3 Exploração dos Frameworks DBI

Diversas técnicas de evasão de *frameworks* de instrumentação binária dinâmica baseiam-se na busca por artefatos no espaço de memória do processo sob análise, devido as mudanças inseridas pelas soluções DBI. Assim, para garantir a compreensão das técnicas apresentadas nos Capítulo seguintes, esta subseção apresenta os conceitos relacionados à estrutura e ao funcionamento da memória de um processo especificamente quando um framework DBI está presente. Referências à memória de processo neste trabalho são sempre a memória virtual alocada pelo sistema operacional e não a memória física. Vale ressaltar que está explicação considera um sistema operacional Windows, principal alvo das técnicas anti-instrumentação introduzidas pela literatura (Falcón and Riva, 2012a; Rodriguez et al., 2016; Sun et al., 2016; Hron and Jermář, 2014).

A Microsoft (Microsoft, 2017b) define o espaço de endereço virtual para um processo como um conjunto de endereços de memória virtual reservados para o uso do mesmo, não sendo possível o acesso direto de um processo ao espaço de endereços de outro. A Figura 2.2, retirada de (Ligh et al., 2014), ilustra como é estruturado o espaço de endereços virtuais no sistema operacional Windows em uma arquitetura 64 bits.

Figura 2.2: Estrutura da memória de um processo (Ligh et al., 2014).

Nota-se na Figura 2.2 a existência de diversas seções - conjuntos de páginas de memória virtual - alocadas em diferentes endereços, com diferentes tamanhos (que não são estáticos) como, por exemplo, as seções que são utilizadas como pilha para as *threads* do processo e a seção que armazena o código executável do processo. O processo tem disponível para seu uso do endereço 0x0 até o 0x7FFFFFFFFFF, totalizando 8 terabytes (8,192 gigabytes) de espaço. Os endereços de 0x7FFFFFFFFFF a 0xFFFFFFFFFFF são reservados para o Sistema Operacional. Cada região em cinza, chamada de segmento de memória, pode ser composta por apenas uma página de memória ou por um conjunto delas. Uma página de memória virtual é a menor unidade de dados do espaço de endereçamento virtual para o gerenciamento de memória em um sistema operacional, segundo Tanenbaum (2008).

Analogamente, também é possível representar um espaço de endereços virtuais quando a aplicação está sob análise de um *framework* de instrumentação binária dinâmica, conforme visto na Figura 2.3.

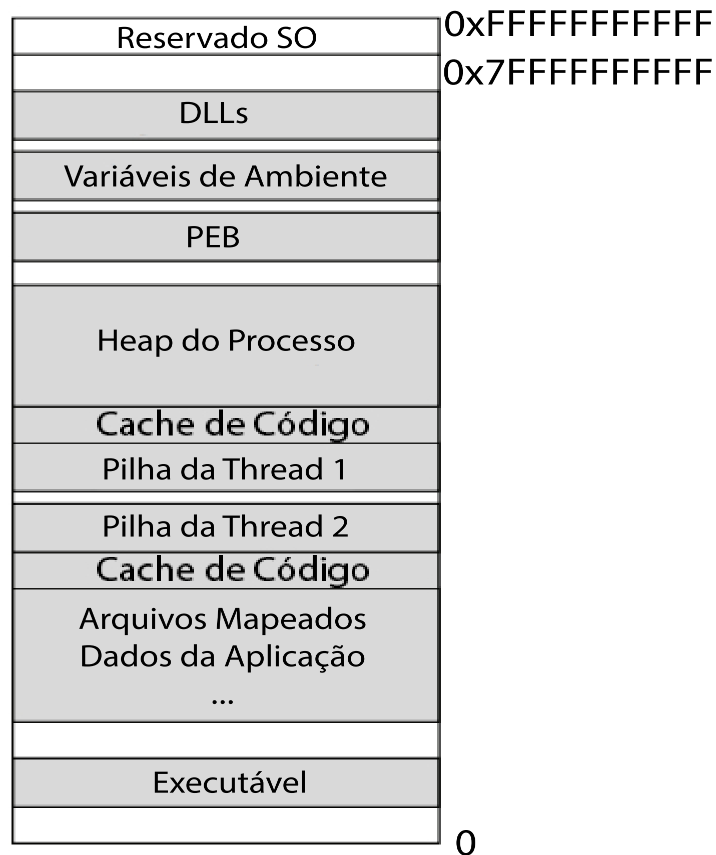

Figura 2.3: Estrutura da memória de um processo (Ligh et al., 2014) modificada para representar a presença de DBI.

Percebe-se na Figura 2.3 que a estrutura básica do espaço de endereços do processo

é mantida. Contudo, nota-se a presença de novas seções, os Caches de Código, que são essenciais para o funcionamento das soluções DBI, pois é neles que reside o código da aplicação que de fato será executado e informações de controle específicas inerentes a instrumentação, como a cópia dos registradores do processador. O número de Caches de Código varia no tempo de acordo com as necessidades dos *frameworks* DBI e com o código da aplicação sendo analisada. O tamanho das Caches de Código pode ser configurado no *frameworks* de instrumentação antes da sua execução. Outras estruturas, menos visuais, são inseridas pelos *frameworks* DBI no espaço de memória virtual do processo, como os ganhos nas bibliotecas do sistema operacional (*DLL Hooks*), apresentados na seção 3.2.3.1.

Alguns artefatos inseridos pelos *frameworks* DBI aumentam a superfície de ataque explorável por aplicações maliciosos, permitindo não apenas a detecção da ferramenta DBI, mas também a fuga e a alteração no fluxo de execução do processo de instrumentação. Como consequência da fuga da instrumentação e subversão do fluxo de execução da ferramenta de instrumentação, a aplicação cliente pode executar códigos arbitrários fora do processo de análise do *framework* DBI, o que é conhecido como ataque de execução de códigos arbitrários (*Arbitrary Code Execution Vulnerabilities*) e a falha na detecção de aplicações maliciosas. Além disso, a execução de códigos arbitrários, mesmo em ambientes controlados, pode levar a comportamentos maliciosos pelo ambiente de análise e *framework* DBI, infecção do sistema por *malwares* e até o comprometimento do computador executando o *framework* de instrumentação.

Sun et al. (2016) definiram seis (06) critérios para medir e determinar quando uma aplicação realiza a fuga do processo de instrumentação binária dinâmica:

1. Quando é capaz de executar instruções proibidas pela ferramenta DBI;
2. Quando executa instruções controladas com contexto DBI controlado;
3. Quando executa instruções controladas operando sobre a pilha do *framework* DBI;
4. Quando executa instruções controladas no contexto crítico da ferramenta DBI;
5. Quando executa instruções controladas alterando o fluxo de controle do *framework* DBI;
6. Quando executa instruções controladas adulterando a aplicação cliente da instrumentação.

Sun et al. (2016) utilizam o termo instruções controladas para se referir ao caso em que a aplicação sob análise tem controle da instrução a ser executada. Esse critérios foram utilizados como base para analisar as técnicas anti-instrumentação presentes na literatura, a fim de permitir a identificação das técnicas evasivas que podem levar à fuga da instrumentação binária dinâmica, através da exploração da superfície de ataque inserida pelas ferramentas DBI.

Posteriormente, Zhechev (2018) atribui a possibilidade de uma aplicação sendo analisada por uma ferramenta DBI realizar a fuga da instrumentação binária dinâmica a duas propriedades dos *frameworks* DBI. A primeira é a reutilização de códigos já compilados anteriormente pelo compilador JIT (*just-in-time*) dos *frameworks* DBI. A segunda é a ausência de verificações de integridade nos Caches de Código. Segundo o autor, os *frameworks* DBI mantém essas características para evitar degradações de desempenho durante a execução das aplicações sendo analisadas.

As principais técnicas para alcançar a fuga da instrumentação e subversão do fluxo de execução são baseadas na manipulação de estruturas críticas do *framework* de instrumentação, como Cache de Código, pilha e *callbacks* aplicação-*framework*. Diferentes técnicas de evasão (por exemplo, *Detecção por Thread Local Storage*, Contexto Inesperado e *Detecção de Code Caches*) serão discutidas no Capítulo 3.

2.3.1 Cache de Código

Como forma de ilustrar uma dessas técnicas e deixar o leitor mais familiarizado com o tema, a Figura 2.4 ilustra o conteúdo de um Cache de Código (Figura 2.4.b) para o *framework* DBI Pin, em relação ao código original da aplicação sem instrumentação (Figura 2.4.a).

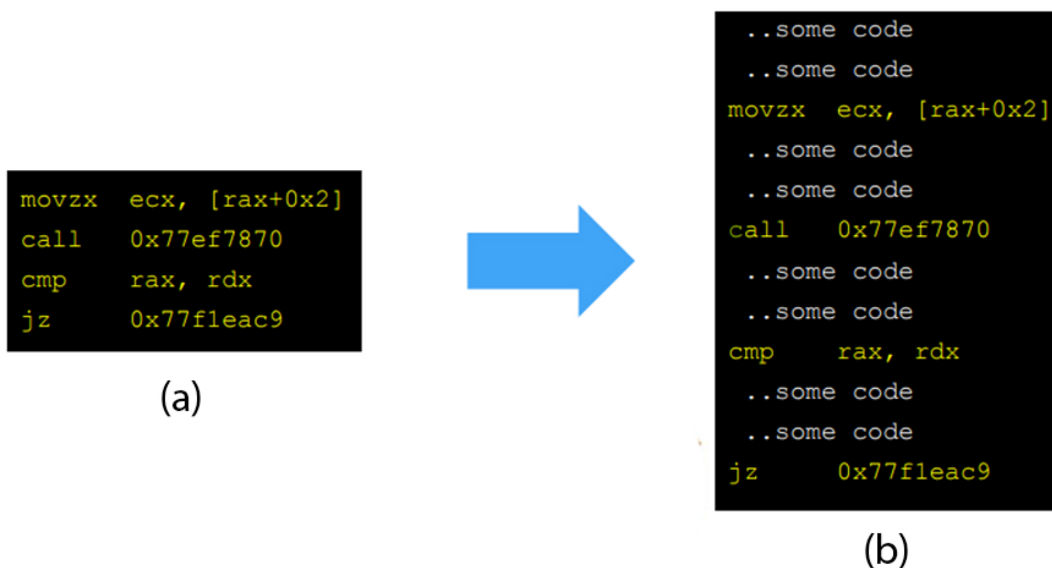

Figura 2.4: Ilustração do conteúdo de um Cache de Código em relação ao código original (Intel Corporation, 2013).

Na Figura 2.4.b, o Cache de Código contém o código com instrumentação de granularidade por instrução inerente às rotinas de análise e ao controle do Pin, ou seja, instruções que garantem que após a execução do código cliente, o controle do fluxo de execução retorne ao *framework* DBI. Nesse cenário, caso a aplicação sob análise do *fra-*

mework DBI possa determinar o endereço da Cache de Código e relacioná-la ao código original é possível reescrever as instruções de controle do Pin com instruções arbitrárias e causar a execução desse trecho de código fora do contexto de análise, subvertendo o fluxo de execução da ferramenta DBI, concretizando a fuga da instrumentação.

Não obstante, é possível ao invés de reescrever as intruções de controle do Pin na Cache de Código, reescrever o código da aplicação com instruções de desvio, redirecionando o fluxo de execução para outra região de memória. O impacto dessa abordagem é permitir a fuga da instrumentação com um número mínimo de alterações na Cache de Código, tornando possível realizar a fuga da instrumentação e posteriormente o retorno para a execução normal do *framework* de instrumentação binária dinâmica. Desta maneira, a fuga da instrumentação torna-se mais transparente, uma vez que o fluxo de execução normal da ferramenta DBI é retomado.

Dentre os mecanismos exploráveis que podem levar a fuga da instrumentação binária dinâmica, os Caches de Código são os que permitem os ataques mais potentes, segundo Zhechev (2018), pois permitem que o atacante espefique livremente qual código será executado fora do contexto de análise da ferramenta DBI (*Arbitrary Code Execution Vulnerabilities*). Sun et al. (2016) descreve um algoritmo de exploração dos Caches de Código, conhecido como *Run, Modify, Run*, através dos seguintes passos:

1. Executar um trecho de código conhecido pela aplicação cliente, a fim de que o *framework* DBI crie um Cache de Código referente a esse código;
2. Encontrar o Cache de Código criado pela ferramenta DBI relativo ao trecho executado;
3. Reescrever o Cache de Código com
   (a) o código que se deseja executar fora do contexto de análise;
   (b) uma instrução de salto de execução para uma região de código que se deseja executar fora do contexto de análise.
4. Executar novamente o trecho de código utilizado como base pela ferramenta DBI para a criação do Cache de Código.

O processo de fuga de instrumentação através da exploração dos Caches de Código inicia com a execução de um trecho de código conhecido pela aplicação, escolhido arbitrariamente. Esse processo leva a ferramenta DBI a interceptar a tentativa de execução e criar um Cache de Código com as instruções que serão de fato executadas, uma vez que o código original da aplicação nunca é executado. O passo seguinte é a obtenção do endereço do Cache de Código criado pelo *framework* DBI.

O rastreamento de Caches de Código pode ser feito através da inspeção da memória virtual do processo, por força bruta ou através de funcionalidades de gerenciamento de memória virtual do sistema operacional. No caso dos sistemas Windows, é possível utilizar APIs (*Application Programming Interface*) como a *VirtualQuery* (Microsoft, 2018j). Em seguida, inicia-se o processo de alteração do Cache de Código selecionado,

onde é possível adotar duas abordagens diferentes. Na primeira (3.a), reescreve-se o Cache de Código, ou parte dele, com as instruções a serem executadas fora da supervisão do *framework* DBI. A segunda abordagem (3.b) é mais poderosa, pois permite com maior facilidade que um código seja executado fora do contexto de análise e em seguida retorne a execução para o código sob análise do *framework* DBI, garantindo transparência da fuga. A modificação de um Cache de Código só é possível porque as ferramentas DBI alocam os segmentos de memória utilizados como Caches de Código com permissões de leitura, escrita e execução (RWX).

Por fim, a aplicação cliente executa novamente o mesmo trecho de código do passo inicial (1). Devido ao reaproveitamento de código, o *framework* DBI irá determinar a execução do Cache de Código previamente criado e que foi modificado pela aplicação, terceiro passo (3).

2.4 Malwares Evasivos

Aplicações maliciosas capazes de detectar quando estão sob análise e tomar ações com base nisso são conhecidas como *malwares* cientes de análise (*malwares analysis-aware* (Rodriguez et al., 2016; Balzarotti et al., 2010)), com dupla personalidade (*split personality malware*(Rodriguez et al., 2016; Balzarotti et al., 2010)) ou evasivos (Polino et al., 2017; Ekenstein and Norrestam, 2017). A incorporação de técnicas evasivas permite que o *malware* verifique onde está sendo executado e se comporte de maneira benigna quando um ambiente de análise é detectado. Como consequência, uma aplicação maliciosa identificada erroneamente como benigna pode se disseminar e penetrar facilmente em um sistema alvo (Carpenter et al., 2007). Segundo McAfee Labs (2017), o *malware* conhecido como *Cascade* foi o primeiro a fazer uso de técnicas de evasão, desenvolvido nos anos 80.

De acordo com Rodriguez et al. (2016) e Balzarotti et al. (2010), os termos *malwares* conscientes de análise (*analysis-aware malware*) e *malwares* com dupla personalidade (*split personality malware*) são equivalentes, sendo definidos como *malwares* capazes de reconhecer quando estão sendo executados em ambientes sob análise e alterar seu próprio comportamento, deixando de executar ações maliciosas, a fim de não serem identificados como *malwares*. Polino et al. (2017) definem *malwares* evasivos como sendo aplicações maliciosas munidas de técnicas, como *anti-debugging* e anti-instrumentação, que permitem esconder seus comportamentos maliciosos quando verificado que estão sendo executados em um ambiente controlado. Assim, segundo os autores, os três termos são equivalentes.

No entanto, autores como Ekenstein and Norrestam (2017) adotam uma definição ligeiramente mais ampla para *malwares* evasivos, os definindo como: uma aplicação maliciosa equipada com técnicas evasivas que permitem evitar a análise do *malware* em um ambiente controlado. Apesar da similaridade entre as definições, a feita por Ekenstein and Norrestam (2017) abrange técnicas evasivas que não requerem a detecção do ambiente de análise para realizar a evasão, o que se dá através de técnicas que exploram limitações no funcionamento das ferramentas de análise, como o uso de

comportamentos (instruções) não suportados pela ferramenta de análise, verificação do padrão de interação do usuário com o sistema e o uso de *stalling code* (Kolbitsch et al., 2011) que atrasa a execução dos comportamentos maliciosos do malware. Essas técnicas são conhecidas como Evasão Não Condicional (*Non-Conditional Evasion* (Bulazel and Yener, 2017)), Técnicas Independentes de Detecção (*Detection-Independent* (Afianian et al., 2018)) ou Técnicas de Evasão Indiretas (Rodriguez et al., 2016). Neste trabalho, essas técnicas são abordadas na seção 3.3.

Segundo Bulazel and Yener (2017), diversos trabalhos propuseram meios de mitigação contra as técnicas de evasão de ambientes de análise, empregando uma grande variedade de soluções, dinâmicas e estáticas, como modificações binárias, melhorias na transparência dos ambientes de análise, exploração de múltiplos fluxos de execução, gravação e *replay* (onde um fluxo de execução é repetido) multiplataforma e análise em *bare metal* (sem virtualização ou emulação). O Capítulo 4 apresenta as soluções existentes para evitar a evasão das ferramentas de instrumentação binária dinâmica.

2.5 Histórico da Evasão dos *Frameworks* DBI

Esta seção apresenta de maneira cronológica os trabalhos relacionados às técnicas de detecção e evasão de ferramentas de instrumentação binária dinâmica, sendo abordados especificamente as técnicas de evasão e as ferramentas que foram propostas como contramedidas, a fim de situar o leitor quanto ao desenvolvimento em quantidade e em sofisticação dos mecanismos de evasão e mitigação. Majoritariamente esses trabalhos são provenientes de conferências de segurança relacionadas à indústria.

Falcón and Riva (2012a) introduziram o primeiro conjunto de técnicas evasivas com o objetivo de detectar o *framework* de instrumentação Intel Pin no sistema operacional (SO) Windows. Os autores apresentaram treze (13) técnicas, com as provas de conceito (PoCs) correspondentes, em uma ferramenta em forma de benchmark, chamada `eXait`. Diversos trabalhos posteriores (Deng et al., 2013; Polino et al., 2017) que propuseram mecanismos de mitigação utilizaram essa ferramenta para verificar a eficácia de suas soluções.

Dois anos depois, Ligh et al. (2014) mostraram que outro *framework* de instrumentação binária dinâmica, o DynamoRIO (Bruening et al., 2001), poderia ser detectado pela aplicação sendo analisada, tanto em ambientes Windows, quanto em ambientes Linux. Ademais, os autores fizeram uma discussão sobre os casos onde a transparência dos *frameworks* DBI é preservada e apontaram possíveis soluções para garantir essa transparência. Similarmente, Hron and Jermář (2014) propuseram seis (06) novas técnicas de evasão contra os *frameworks* DBI Pin e DynamoRIO baseadas no uso códigos automodificantes e em como esses *frameworks* DBI gerenciam a memória virtual da aplicação cliente. Os autores apresentaram também provas de conceito para cada técnica. Sun et al. (2016) apresentaram seis (06) novas técnicas evasivas contra os *frameworks* DBI Pin e DynamoRIO. Eles introduziram os conceitos de fuga da instrumentação binária dinâmica e comprometimento do ambiente de análise DBI, além de desenvolverem uma forma de medir a fuga da instrumentação e abordagens para concretizá-la.

Quanto aos trabalhos acadêmicos existentes, Rodriguez et al. (2016) foram os pioneiros ao realizar uma revisão das técnicas anti-instrumentação existentes até aquele momento, propor uma taxonomia e apresentar um conjunto de contramedidas para impedir a detecção das ferramentas DBI. Os autores agruparam o conjunto de contramedidas contra as técnicas anti-instrumentação em um ferramentas extensível baseada no *framework* DBI Pin, chamada `PinVMShield`. Posteriormente, Polino et al. (2017) apresentaram uma revisão e classificação de um conjunto de técnicas anti-DBI focadas no *framework* Pin. Os autores sugeriram contramedidas para cada técnica de evasão apresentada e as agruparam em uma ferramenta nomeada como `Arancino`. Eles também validaram a eficácia dessas contramedidas através da ferramenta `eXait` (Falcón and Riva, 2012a).

Recentemente, Zhechev (2018) publicou uma tese de mestrado focada no questionamento se *frameworks* de instrumentação binária dinâmica são ferramentas apropriadas para analisar programas desconhecidos e/ou potencialmente maliciosos. Além de demonstrá-la, o autor também realizou uma discussão sobre a fuga da instrumentação e sobre os motivos que a tornam possível, sendo o principal: o modelo de memória compartilhada adotado pelos *frameworks* de instrumentação binária dinâmica. De acordo com Zhechev, os *frameworks* DBI não seriam apropriados para a análise de programas evasivos, por não serem capazes de garantir as características isolamento e transparência, e por levarem ao aumento da superfície de ataque. Por fim, o autor apresentou treze (13) novas técnicas de evasão contra o *framework* DBI Pin em ambientes Linux. O Capítulo seguinte irá abordar em detalhes essas técnicas de evasão e as demais propostas na literatura, que agem contra os *frameworks* de instrumentação binária dinâmica.

Capítulo 3

Técnicas de Evasão

Este Capítulo descreve trabalhos que apresentam técnicas anti-instrumentação introduzidas pela literatura, apresentada na subseção 2.5, que podem ser utilizadas por aplicações maliciosas para detectar a presença dos *frameworks* DBI e evadí-los. A seção 3.1 apresenta as classificações (ou taxonomias) feitas por diferentes autores (Rodriguez et al., 2016; Sun et al., 2016; Polino et al., 2017; Zhechev, 2018) como forma de agrupar as técnicas de evasão DBI. Uma discussão sobre as limitações de cada taxonomia é feita e a proposição de uma nova classificação capaz de agrupar as técnicas atuais em duas vertentes (direta e indireta) é apresentada. Em seguida, as seções 3.2 e 3.3 discutem todas as técnicas diretas e indiretas, respectivamente. Por fim, uma discussão sobre todos os trabalhos é feita na seção 3.4.

3.1 Classificação das Técnicas Anti-Instrumentação

A primeira classificação das técnicas de evasão de *frameworks* DBI foi proposta por Rodriguez et al. (2016), especificamente para agrupar as técnicas anti-instrumentação focadas no *framework* Pin. Os autores classificaram as técnicas de evasão DBI de duas formas. A primeira é a indireta, quando a técnica de evasão não incorpora artefatos de código com o objetivo de detectar o *framework* DBI. Assim, a evasão ocorre sem a necessidade de detectar o *framework*. A segunda é a direta, cujo objetivo é detectar a presença do *framework* DBI para evadí-lo. As técnicas de evasão diretas foram subdivididas em seis (06) categorias:

1. **Detecção da Máquina Virtual do Pin**: Agrupa as técnicas de evasão que focam em reconhecer a presença da biblioteca da máquina virtual do Pin, conhecida como *pinvm.dll*.

2. **Detecção de Canais de Comunicação**: Conjunto de técnicas anti-instrumentação que detectam os mecanismos de comunicação inter-processo utilizados pelo *framework* Pin.

3. **Detecção por Variações de Tempo**: Reúne as técnicas que apontam a presença do *framework* DBI através das variações no tempo de execução da aplicação cliente inseridas pelo processo de instrumentação binária dinâmica.

4. **Detecção do Compilador JIT do PIN**: É o conjunto composto pelas técnicas de evasão de *framework* que identificam a presença do compilador JIT (*just-in-time*) do Pin, através dos artefatos deixados por ele no proesso da aplicação cliente.

5. **Detecção Pelo Valor do Registrador de Próxima Instrução (EIP)**: Agrupa as técnicas que apontam a presença do *framework* DBI Pin através da monitoração e validação do conteúdo assumido pelo registrador EIP durante a execução da aplicação sob análise.

6. **Detecção Por Outras Técnicas**: Agrega as técnicas de evasão que não se encaixam em nenhuma das outras categorias e revelam a presença de artefatos que podem ser utilizados para apontar o *framework* de instrumentação Pin.

A classificação seguinte foi proposto por Sun et al. (2016) ao agrupar as técnicas de evasão para os *frameworks* Pin e DynamoRIO em duas categorias: (i) **Detecção Passiva com Código de Bloqueio**, que compreendem técnicas anti-instrumentação que permitem a evasão das análises realizadas pelos *frameworks* DBI sem a necessidade de detectá-los; (ii) **Detecção Ativa**, que compreendem técnicas que focam em detectar os *frameworks* de instrumentação, para assim evadí-los.

Depois, Polino et al. (2017) propuseram uma classificação considerando as técnicas de evasão DBI em quatro (04) categorias:

1. **Artefatos de Cache de Código**: Compreende técnicas que detectam se o código está sendo executado a partir de uma cache de código.

2. **Artefatos de Ambiente**: Agrupa as técnicas que detectam o *framework* DBI a partir dos artefatos deixados pelo processo de instrumentação no sistema, durante a execução.

3. **Detecção do compilador JIT**: Técnicas focadas em detectar o compilador JIT (*just-in-time*) do *framework* de instrumentação.

4. **Detecção de *Overhead***: Técnicas que determinam se o código está sob instrumentação a partir de métricas de desempenho da execução do cliente, como tempo de execução.

Finalmente, Zhechev (2018) propôs três (3) categorias para agrupar as técnicas de evasão de acordo com o componente alvejado do *framework* DBI:

1. **Cache de Código / Artefatos de Instrumentação (CA)**: Agrupa as técnicas anti-instrumentação que detectam as anomalias introduzidas pelo fato do código sendo executado não ser o original da aplicação cliente.

2. ***Overhead* do Compilador JIT (CO)**: Compreende as técnicas de evasão que detectam o *overhead* temporal introduzido pelo uso do compilador JIT empregado pelas ferramentas DBI.

3. **Artefatos do Ambiente de Tempo de Execução (EA)**: Técnicas que exploram a presença dos artefatos de ambiente introduzidos pelos *frameworks* DBI.

As classificações propostas por Rodriguez et al. (2016) e Sun et al. (2016) agrupam as técnicas de evasão utilizando duas categorias principais que são equivalentes, técnicas de evasão diretas e técnicas de detecção ativa, respectivamente. Ambas agrupam as técnicas de evasão focadas em detectar o *framework* DBI, para assim evadí-lo. Da mesma forma, as categorias técnicas de evasão indiretas (Rodriguez et al., 2016) e detecção passiva com código de bloqueio (Sun et al., 2016) reúnem as técnicas que evadem os *frameworks* de instrumentação sem a necessidade de detectá-los.

Diferente de Sun et al. (2016), Rodriguez et al. (2016) expandem a categoria de técnicas de evasão diretas com seis (06) subdivisões. Essas subdivisões são similares às classificações propostas por Polino et al. (2017) e Zhechev (2018) e agrupam as técnicas de evasão sob critérios semelhantes. As categorias *Detecção da máquina virtual do Pin*, *Detecção de canais de comunicação* e *Detecção por outras técnicas* de Rodriguez et al. (2016) estão contidas nas categorias *Artefatos de Ambientes* e *Detecção do Compilador JIT* de Polino et al. (2017) e, correspondentemente, em *Artefatos de Ambiente* de Zhechev (2018), sendo conceitualmente equivalentes. Já a categoria *Detecção Pelo Valor do Registrador de Próxima Instrução (EIP)*, de Rodriguez et al. (2016), está contida nas categorias *Artefatos de Cache de Código* de Polino et al. (2017) e *Cache de Código/Artefatos de Instrumentação* de Zhechev (2018). Por fim, as categorias *Detecção Por Variações de Tempo*, *Detecção de Overhead* e *Overhead do Compilador JIT* de Rodriguez et al. (2016), Polino et al. (2017) e Zhechev (2018) são equivalentes.

Apesar das similaridades, a taxonomia proposta por Rodriguez et al. (2016) visa agrupar técnicas de evasão focadas no *framework*de instrumentação Pin, enquanto as de Polino et al. (2017) e Zhechev (2018) são mais genérica, se tornando independentes da ferramenta DBI em questão. No entanto, Polino et al. (2017) e Zhechev (2018) não consideram e existências das técnicas de evasão indiretas, tornando suas taxonomias incapazes de agrupar uma série de técnicas anti-instrumentação.

Desta forma, considerando as taxonomias apresentadas, é possível perceber que nenhuma delas é capaz de agrupar as técnicas de evasão de maneira independente do *framework* DBI alvejado e considerando as naturezas direta e indireta, também conhecida como não condicional (Bulazel and Yener, 2017), das técnicas anti-instrumentação. Portanto, nesta dissertação é proposta uma taxonomia que combina as classificações apresentadas, de forma que seja possível agrupar as técnicas de evasão presentes na literatura. Para tanto, será adotada a divisão proposta por Rodriguez et al. (2016) em técnicas de evasão diretas e indiretas, onde as técnicas de evasão diretas serão subdivididas utilizando a classificação proposta por Polino et al. (2017), uma vez que é independente de *frameworks* DBI específicos. Além disso, este trabalho propõe ainda duas novas classes de técnicas de evasão indiretas na taxonomia, que são baseadas na

limitação explorada pela técnica evasiva. A Figura 3.1 ilustra a classificação resultante proposta neste trabalho para agrupar as técnicas de evasão.

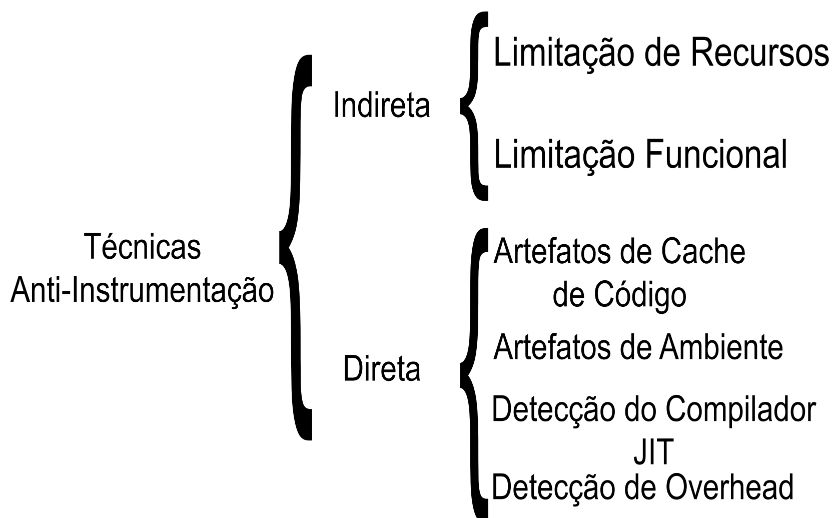

Figura 3.1: Classificação resultante para as técnicas de evasão

As seções seguintes apresentam, detalhadamente, as técnicas de evasão dos *frameworks* DBI presentes na literatura, discutindo seus mecanismos de funcionamento e validando suas eficácias quando disponíveis provas de conceito, enumerado na Tabela 3.1.

3.2 Técnicas Diretas de Evasão de Frameworks DBI

Esta seção apresenta as técnicas de evasão agrupadas de acordo com a classificação apresentada na seção anterior. Primeiramente, as técnicas de evasão diretas são apresentadas seguindo a ordem: Artefatos de Cache de Código, Artefatos de Ambiente, Detecção do Compilador JIT e Detecção de *Overhead*. Em seguida as técnicas anti-instrumentação indiretas são apresentadas.

3.2.1 Artefatos de Cache de Código

Esta subseção apresenta as técnicas anti-instrumentação que exploram artefatos e comportamentos particulares ao uso de Caches de Código pelos *frameworks* DBI. Em geral, as técnicas agrupadas nesta categoria exploram fato do código sendo executado durante a execução da análise DBI não ser o original da aplicação cliente.

3.2.1.1 Detecção de EIP/RIP

Quando uma aplicação está sendo analisada por um *framework* de instrumentação binária dinâmica, o código em execução é proveniente de regiões de memória conhecidas como caches de código, ao invés da região de memória (segmento) onde o executável foi carregado. Essa diferença nos endereços das instruções sendo executadas é observável no valor do registrador *Instruction Pointer* (EIP), responsável por armazenar o endereço virtual da próxima instrução a ser executada. Segundo o manual da Intel Corporation (2011), o registrador EIP pode ser referenciado também como *Relative Instruction Pointer* (RIP) para sistemas de 64 bits

Em razão dos *frameworks* DBI utilizarem o mecanismo cache de código, o código original da aplicação a ser instrumentada não chega a ser executado. Desse modo, diversos trabalhos (Falcón and Riva, 2012a; Hron and Jermář, 2014; Rodriguez et al., 2016; Polino et al., 2017; Zhechev, 2018) sugerem que a aplicação cliente pode detectar, através do valor do registrador EIP, que o código sendo executado não está na mesma região de memória do código original da aplicação. Existem instruções que carregam em memória o conteúdo dos registradores, incluindo o EIP, como *FSTENV*, *FSAVE* e *FXSAVE*.

3.2.1.2 Código Automodificante

Uma vez que o código original da aplicação não é executado, mas sim uma cópia dele que reside no Cache de Código, o instrumentador binário dinâmico precisa ser capaz de lidar com os códigos automodificantes (*Self-Modifying Code* - SMC), isto é, códigos que alteram suas instruções, sobrescrevendo-as em tempo de execução (Polino et al., 2017). Segundo Hron and Jermář (2014) e Polino et al. (2017), a aplicação cliente pode detectar caso o código automodificante não seja corretamente tratado, podendo até encerrar sua execução abruptamente.

De acordo com Hron and Jermář (2014), duas outras técnicas de evasão podem fazer uso do código automodificante para detectar a presença da instrumentação binária dinâmica: (1) *Block Cache vs. Virtual Memory State* (estados da cache de código vs. memória virtual) e (2) *Inherent weakness of the write-protecting appro-ach to SMC* (fraquezas inerentes à proteção de escrita quando ocorre o código automodificante).

No que diz respeito ao **estados da Cache de Código vs Memória Virtual**, a Cache de Código possui a característica de sempre ter as permissões de leitura, escrita e execução. Caso o código cliente tente mudar as permissões da memória da região onde reside seu código, utilizando SMC, o instrumentador binário dinâmico precisa garantir que, para a aplicação, as permissões realmente tenham sido alteradas. Caso contrário, o cliente poderá detectar a instrumentação binária dinâmica.

Ja sobre **fraquezas inerentes à proteção de escrita quando ocorre código automodificante**, caso haja uma tentativa de escrita por parte de um código automodificante em uma região com proteção de memória à escrita, ocorrerá uma exceção que, se tratada, desviará o fluxo de execução para outra região na memória e alterará para outro estado a pilha do processo. Se um dado for deixado na pilha, na região marcada

como não utilizada e ocorrer a exceção descrita, o instrumentador binário dinâmico precisará garantir que esse dado permanecerá na pilha, na região marcada como não utilizada. De outro modo, a aplicação cliente poderá detectar que está sob análise.

3.2.1.3 Contexto Inesperado

Segundo Sun et al. (2016), o *framework* Pin mantém uma cópia (*backup*) de todos os registradores do sistema em uma região memória de Cache de Código. Além disso, o Pin reserva o registrador EBX apenas para seu uso, empregando-o para armazenar o endereço base dos backups dos registradores. Quando as intruções da aplicação cliente referenciam, ou utilizam o registrador EBX, o Pin toma a responsabilidade por reescrevê-las.

Sun et al. (2016) mostraram que caso a aplicação cliente localize na memória os backups dos registradores, é possível alterar a cópia do registrador *EBX* utilizado pelo Pin. Essa mudança reflete no conteúdo real do registrador EBX, o que pode ser utilizado para revelar a presença do *framework* DBI.

3.2.1.4 Exceções Inesperadas

Uma vez que os *frameworks* DBI alocam o código da aplicação cliente no Cache de Código conforme demandado pela execução do programa sob análise, podem ocorrer comportamentos não esperados em aplicações que realizam cálculos de *checksum* do código em execução - um valor calculado a partir das instruções em execução, usado para verificar a integridade do código. Segundo Ligh et al. (2014), o *framework* DBI DynamoRIO não é capaz de lidar corretamente com aplicações que empregam *checksum*, aprensentando exceções de violação de acesso a memória.

3.2.1.5 *Xmode Code*

A técnica de evasão *Xmode Code* explora a funcionalidade dos sistemas operacionais Windows, WoW64, que é um emulador que permite a execução de códigos Windows 32 bits em ambientes 64 bits de forma transparente Microsoft (2018h), permitindo inclusive que uma mesma aplicação execute ambos tipos de código. De forma prática, quando uma aplicação projetada para sistemas 64 bits está em execução, o registrador CS possui o valor hexadecimal 0x33, indicando que o código está sendo executado em *64-bit mode*. Quando uma aplicação com código 32 bits está em execução em um sistema 64 bits, através do WoW64, o registrador CS possui o valor hexadecimal 0x23, indicando que está em *compatibility mode*.

No entanto, segundo Sun et al. (2016), quando uma aplicação está sob análise do *framework* DynamoRIO, o registrador CS fica fixo em *compatibility mode*, com o valor hexadecimal 0x23, não sendo possível fazer a mudança para 64-bit mode. Esse comportamento pode ser utilizado pela aplicação cliente para detectar a presença do *framework*. Caso haja a tentativa de execução de um código 64 bits incompatível com o modo 32 bits, com o registrador CS em *compatibility mode*, pode acontecer o

encerramento abrupto da execução do *framework* de instrumentação e da aplicação cliente, como descrito na seção 3.3.4.

3.2.1.6 Abuso das Instruções de *syscall*

Segundo Zhechev (2018), quando uma chamada de sistema (*system call*) é executada através da instrução *syscall*, o registrador *rcx* é sobrescrito com o conteúdo do registrador *instruction pointer* (EIP/RIP), o que permite ao sistema operacional retomar a execução normal do programa após a *system call*, através da instrução *sysret*. No entanto, quando uma aplicação está sendo analisada pelo *framework* Pin, esse comportamento não é observado. O registrador *rcx* não é alterado pela chamada de sistema, se mantendo estático. Esse comportamento é observável pela aplicação cliente e pode ser utilizado para indicar que uma ferramenta DBI está em execução.

3.2.1.7 Emulação incorreta da instrução *rdfsbase*

A instrução *rdfsbase* permite que o conteúdo dos registradores de segmento (*segment registers*) FS e GS sejam carregados em um registrador apontado como destino na instrução, nos modelos mais recentes dos processadores Intel CPUs (Ivy Bridge e superiores), como alternativa ao uso da chamada de sistema *SYS_arch_prctl*. O *framework* Pin falha ao emular a execução da instrução *rdfsbase*, retornando como resultado os valores dos *segment registers FS and GS* do *framework* DBI, ao invés dos da aplicação. Enquanto isso, a *system call SYS_arch_prctl* é tratada corretamente, retornando os os valores dos *segment registers FS and GS* da aplicação. Essa discrepância dos valores retornados pode ser utilizada como indicativo da presença do Pin. Essa técnica foi introduzida por Zhechev (2018) para ambientes Linux.

3.2.1.8 Negligenciamento do No-eXecute Bit (nx)

Os fabricantes de processadores desenvolveram as tecnologias *No-eXecute Bit (nx)*, AMD, e *Execute Disable (XD)*, Intel, que permitem que os sistemas operacionais marquem páginas de memória como executáveis, ou não, como uma medida de segurança. Nos SOs Linux, o mecanismos que faz uso dos bits nx, ou XD, é conhecido como X $\oplus$ R. Segundo Zhechev (2018), os *framework* DBI Pin e DynamoRIO negligenciam a marcação X $\oplus$ R fornecida pelo sistema operacional. Assim, o que ocorre na prática é que regiões de memória que não poderiam ser executadas, podem ser executadas quando uma aplicação está sob a análise de uma ferramenta DBI.

3.2.2 Artefatos de Ambiente

Os artefatos de ambiente são as informações mais utilizadas para detectar a presença de *frameworks* DBI. Dentre as 29 técnicas de evasão, 15 utilizam artefatos do ambiente, uma vez que os *frameworks* DBI trazem grandes modificações no processo de execução da aplicação cliente, desde alterações no espaço de memória do processo até alterações em mecanismos de sincronismo e comunicação de *threads*.

3.2.2.1 Detecção do Processo Pai

Quando uma aplicação está sendo instrumentada, é executada como um processo filho do *framework* DBI. Assim, a aplicação cliente pode verificar o nome do seu processo pai no sistema operacional a fim de determinar se é um instrumentador ou não. É uma técnica de detecção de *frameworks* DBI que foi inicialmente proposta por Falcón and Riva (2012a) para detectar o Pin. Posteriormente, Ligh et al. (2014) demonstraram que é uma técnica eficaz contra o DynamoRIO também. Diversos trabalhos (Ligh et al., 2014; Falcón and Riva, 2012a; Rodriguez et al., 2016; Polino et al., 2017) sugeriram a Detecção do Processo Pai como meio para detectar e evadir os *frameworks* de instrumentação. Em parte, isso se deve por ser uma técnica de implementação simples e agir sobre diferentes*frameworks* de instrumentação.

A Figura 3.2 ilustra a árvore de processos de uma aplicação sob análise do *framework* DBI Pin. É possível notar a ocorrência do processo `pin.exe` como pai e filho do processo sendo analisado, a calculadora do Windows (`calc.exe`). Além disso, nota-se que o processo nó inicial da árvore foi o Console de Comandos do Windows, uma vez que essa é a maneira padrão de iniciar o Pin (Luk et al., 2005b).

cmd.exe	4864	TRAVESTI\nriva
pin.exe	3708	TRAVESTI\nriva
calc.exe	2392	TRAVESTI\nriva
pin.exe	6108	TRAVESTI\nriva

Figura 3.2: Árvore de processos para uma aplicação sob análise do Pin (Falcón and Riva, 2012a).

3.2.2.2 *Memory Fingerprinting* / Detecção de Cache de Código

Como explicado na seção 2.2.1, durante a instrumentação binária dinâmica o código original da aplicação cliente não é executado, mas sim uma cópia que reside no Cache de Código. Sun et al. (2016) demonstraram que essa cópia pode ser encontrada por meio de uma varredura na memória do processo sob análise. Assim, caso a aplicação cliente detecte duas ou mais ocorrências de um código que deveria ser único na memória, isso apontaria a presença do instrumentador binário dinâmico.

Segundo os autores, esse processo inicia com a execução intencional de uma sequência de instruções que ocorre apenas uma vez durante a execução do programa. Esse passo força o *framework* DBI a criar um Cache de Código referente a essa sequência de instruções, outrora única na memória. Em seguida, todo o espaço de endereços do processo é verificado, a fim de contabilizar o número de ocorrências da sequência de instruções executada inicialmente. Essa técnica de evasão foi nomeada como **Code Cache Detection**, ou Detecção de Cache de Código.

A Figura 3.3 ilustra o processo de varredura de memória quando a aplicação não

está sob análise (parte superior da figura) e quando a aplicação está sob análise do *framework* DBI Pin (lado inferior da figura).

```
sigature function executed.
signature found @ 0x13c1038
memory search completed, signature count 1
```

```
sigature function executed.
signature found @ 0x1161038
signature found @ 0x18478b2
memory search completed, signature count 2

DBI Detected!!

DBI tool = PIN!
```

Figura 3.3: Varredura em memória por assinaturas de Cache de Código (Sun et al., 2016).

Além disso, Sun et al. (2016) sugeriram que os Caches de Código podem ser detectados através da identificação de assinaturas no seu conteúdo, ou seja, artefatos de memória que são característicos nos Caches de Código dos *frameworks* de instrumentação. Um exemplo dessas assinaturas no *framework* Pin é a sequência hexadecimal *0xfeedbeaf* que está presente em regiões de memória utilizadas como Caches de Código (Sun et al., 2016).

Posteriormente, Polino et al. (2017), de maneira análoga a Sun et al. (2016), sugeriram que os *frameworks* DBI poderiam ser detectados através de uma varredura no espaço de endereços do processo sob análise em busca de artefatos de memória relacionados aos *frameworks* DBI, como strings e padrões de códigos. Ou seja, tratando-se de uma expansão do conceito apresentado na técnica apresentada por Sun et al. (2016). Esta técnica foi nomeada como *Memory Fingerprinting* - nesse contexto, *fingerprinting* refere-se a fazer um mapeamento da memória virtual.

3.2.2.3 Detecção Por *Argv*

De acordo com Falcón and Riva (2012a) é possível detectar o *framework* de instrumentação Pin através de uma verificação dos parâmetros de entrada passados (*argv*) ao processo pai de uma aplicação sob análise DBI. Segundo a documentação da Microsoft (2018b), o argumento de linha de comando *argv* é definido como uma matriz de cadeias de caracteres terminadas em nulo que representam os argumentos de linha de comando inseridos pelo usuário do programa.

Frameworks DBI, como o Pin, utilizam argumentos de linha de comando para definir qual será a aplicação alvo de análise e qual será a *pintool* a ser executada. Considerando que a aplicação cliente divide o espaço de endereços com o *framework* de instrumentação, é possível realizar uma varredura na memória do processo em busca dos argumentos passados para a aplicação, a fim de determinar se a aplicação cliente está sob instrumentação e até mesmo qual ferramenta de análise está sendo executada,

através da análise desses argumentos. Rodriguez et al. (2016) corroboraram com Falcón and Riva (2012a), afirmando que a verificação dos parâmetros de entrada passados ao programa (*argv*) poderia ser utilizada para determinar a presença do *framework* de instrumentação Pin.

3.2.2.4 *Handle* de Processo

Segundo Falcón and Riva (2012a), quando uma aplicação está sob análise de um *framework* DBI, como o Pin, é possível atestar a presença do *framework* DBI através de uma enumeração dos *Process Handles* (*handle* de processos) ativos - identificadores únicos de um processo no sistema operacional que são gerados na hora da criação de um novo processo Microsoft (2018g). Para o caso do Pin, uma vez que o processo da aplicação cliente é criado pelo *framework* DBI, este mantêm um *Process Handle* ativo referente a aplicação cliente. Falcón and Riva (2012a) afirmam que é possível a partir da aplicação cliente detectar esse *Process Handle*, revelando a presença do *framework* DBI.

3.2.2.5 Número de *File Handlers*

A quantidade de *File Handlers* ativos - identificadores únicos associados ao processo de abertura de um arquivo, atribuídos pelo sistema operacional Microsoft (2018d) - é maior quando um *framework* DBI está em execução, devido aos *file handlers* utilizados por ele. Ligh et al. (2014) mostraram que uma aplicação é capaz de detectar a presença do *framework* DynamoRIO através da verificação do número de *File Handlers* ativos.

3.2.2.6 Número Máximo de *File Handlers* Abertos

Para os casos em que o sistema operacional permite a alteração no número máximo de *file handlers*, como o Linux, Ligh et al. (2014) demonstraram que o número máximo de *file handlers* que podem ser utilizados em um processo é inferior quando o *framework* DynamoRIO está em execução, em relação a execução nativa da aplicação. No caso apresentado pelos autores, ocorre a redução do *hard limit* (Die.net, 2018) de 4096 file handlers para 4000.

3.2.2.7 Pico no Uso de Memória

Ligh et al. (2014) demonstraram que pico de memória virtual utilizada por uma aplicação, *Peak Memory Usage*, é maior quando o processo está sob análise do *framework* DynamoRIO. Esse aumento é causado pelos artefatos inseridos pelo *framework* DBI no espaço de endereços do processo, como Caches de Código e a biblioteca do DynamoRIO. Assim, caso uma aplicação possa comparar os picos de memória utilizada sob instrumentação e o esperado em modo nativo, poderia ser capaz de detectar a presença do *framework* DBI.

3.2.2.8 Alteração no Número Máximo de *File Handlers*

Segundo Ligh et al. (2014), em sistemas operacionais baseados em Linux, onde é possível alterar o número máximo de *file handlers* que podem ser utilizados por um processo, o framework DBI DynamoRIO não permite que essa modificação ocorra enquanto a aplicação cliente está em execução. Uma das maneiras de alterar o número máximo de *file handlers* é através da utilização da chamada de sistema *setrlimit*, que de acordo com os autores, sempre falha, independente do número máximo de *file handlers* que esteja sendo atribuído. Assim, uma aplicação pode detectar a presença do *framework* de instrumentação caso falhe ao alterar o número máximo de *file handlers*.

3.2.2.9 Detecção pela Máscara de Sinal (*Signal Masks*)

Em sistemas operacionais baseados em Linux é possível verificar quais são os sinais do sistema operacional sendo monitorados por um processo, através da verificação da máscara *SigCgt* Linux Programmer's Manual (2018). Ligh et al. (2014) demonstraram que o *framework* DynamoRIO monitora todos os sinais, enquanto evita modificá-los, através dessa máscara. Assim, é possível para uma aplicação constatar a presença do *framework* DynamoRIO através da verificação da máscara *SigCgt* atribuída a seu processo. A literatura, no entanto, não aponta outros casos de teste contra outros *frameworks* de instrumentação.

3.2.2.10 *Fingerprinting pinvm.dll*/ Detecção de Bibliotecas do DynamoRIO

Frameworks DBI comumente injetam suas bibliotecas no processo sob análise. O Pin, por exemplo, injeta a biblioteca *pinvm.dll* que, como mostrado por Falcón and Riva (2012a), pode ser detectada através de uma varredura na memória virtual do processo por padrões de strings ou mesmo através de uma busca nas funções exportadas pelas bibliotecas, a fim de localizar funções exportadas pela *pinvm.dll*, como *PinWinMain* e *CharmVersionC*. Posteriormente, Rodriguez et al. (2016) reconhecem a técnica *Fingerprinting pinvm.dll* como eficaz para a detecção do *framework* Pin, por uma aplicação sob análise. Como o nome sugere, *Fingerprinting pinvm.dll* é uma técnica focada na evasão do framework de instrumentação Pin, no entanto, Ligh et al. (2014) mostraram que é possível estendê-la a outros *frameworks*, como DynamoRIO.

3.2.2.11 Detecção Pelas Funções Exportadas Por *Pintools*

Como explicado na seção 2.2.1, o *pintool* é o componente lógico do *framework* de instrumentação Pin, contendo a rotina de instrumentação que decide onde e qual código deve ser inserido e pelo código que será inserido, conhecido como rotina de análise. Segundo Falcón and Riva (2012a), existem funções comuns exportadas por todas as *pintools* que podem ser detectadas pela aplicação sob análise, a fim de confirmar a presença do *framework* de instrumentação. Dessa forma, segundo os autores, é possível

atestar a presença do *framework* DBI Pin através da inspeção das funções exportadas pelas bibliotecas.

3.2.2.12 Detecção Pelos Nomes de Seções

Quando uma aplicação está sob análise do *framework* Pin, passam a existir mais seções alocadas no seu espaço de memória virtual. Falcón and Riva (2012a) demonstraram que são seções referentes aos *pintools* e a biblioteca *pinvm.dll* - seções nomeadas como *.pinclie* e *.charmve*, ilustradas na Figura 3.4. Uma aplicação cliente pode realizar uma varredura no seu espaço de memória virtual a fim de detectar essas seções, revelando a presença do *framework* de instrumentação Pin. Posteriormente, Rodriguez et al. (2016) confirmaram a eficácia dessa técnica de evasão contra o Pin.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address
Byte[8]	Dword	Dword	Dword	Dword
.text	002791CC	00001000	00279200	00000400
.rdata	00085DF7	0027B000	00085E00	00279600
.data	0002541C	00301000	00002400	002FF400
.pinclie	00000380	00327000	00000400	00301800
.charmve	00000043	00328000	00000200	00301C00
.reloc	00019878	00329000	00019A00	00301E00

Figura 3.4: Listagem das seções de um processo sob análise do Pin (Falcón and Riva, 2012a).

3.2.2.13 Detecção Por *Event Handles*

Os *Event Objects* são mecanismos de comunicação interprocessos (IPC) do sistema operacional Windows, úteis para enviar sinais a threads de um processo ou mesmo a outros processos, informando a ocorrência de determinado evento do sistema (Microsoft, 2018c). Após a criação de um *Event Object*, ele é manipulado através de *handles*. Segundo Falcón and Riva (2012a), o *framework* Pin faz uso da comunicação interprocessos (IPC) por eventos. Sendo possível, a partir da aplicação cliente listar e verificar os *Event Objects* existentes, o que pode ser utilizado para revelar a presença do *framework* DBI Pin. Rodriguez et al. (2016) corroboraram com Falcón and Riva (2012a) confirmando a eficácia da técnica, acrescentando ainda que os canais de comunicação interprocesso utilizados pelo *framework* Pin possuem normalmente o prefixo *"Pin_IPC"* no nome.

3.2.2.14 Detecção Por *Section Handles*

Shared Section é uma funcionalidade fornecida pelos sistemas operacionais Windows que permite o compartilhamento, entre processos, de porções de memória virtual. É comumente utilizada para a comunicação interprocessos (IPC) (Cerrudo, 2005). Após a criação de um *Shared Section*, ele é manipulado através de *handles*. Falcón and Riva (2012a) mostraram que o *framework* Pin faz uso dessa funcionalidade e que é possível para uma aplicação sob análise do *framework* DBI detectar o uso do *Shared Section*. A posteriori, Rodriguez et al. (2016) corroboraram com Falcón and Riva (2012a), confirmando a eficácia das técnicas de evasão por detecção dos canais de comunicação entre processos utilizados pelo Pin.

3.2.2.15 Detecção por *Thread Local Storage* (TLS)

Segundo o manual da Microsoft (2018i), *Thread Local Storage* (TLS) é uma funcionalidade provida pelos sistemas operacionais que permite que sejam fornecidos dados exclusivos para cada thread de um processo, armazenados em uma estrutura de dados em formato de vetor, com no mínimo 64 posições e no máximo 1.088, acessíveis através de um índice global. Dessa forma, uma thread aloca um índice que pode ser utilizado por outras threads para recuperar os dados exclusivos associados aquela posição do vetor. Ou seja, variáveis TLS podem ser vistas como variáveis globais visíveis apenas para uma thread em particular e não por todo o programa. De acordo com Sun et al. (2016), o *framework* Pin faz uso do TLS e esse comportamento pode ser observável a partir da aplicação cliente. Assim, uma aplicação sob análise do *framework* DBI Pin pode fazer uma verificação dos índices TLS sendo utilizados, revelando a presença do *framework* DBI.

A Figura 3.5 ilustra uma comparação feita por Sun et al. (2016) entre a estrutura TLS sem o Pin (lado esquerdo da figura) e com o Pin (lado direito da figura). No lado direito da Figura é possível notar a presença de duas entradas (1, 2) que são utilizadas pelo *framework* DBI para referenciar Caches de Código.

3.2.2.16 Variáveis de Ambiente Internas do Pin

Segundo Zhechev (2018), para o correto funcionamento do *framework* DBI Pin em ambientes Linux, determinadas variáveis de ambiente precisam estar criadas e com seus devidos valores atribuídos, como *PIN_INJECTOR64_LD_LIBRARY_PATH*, uma variável de ambiente. Devido a aplicação cliente ser executada e analisada como processo filho do *framework*, essas variáveis de ambiente se tornam acessíveis a essa aplicação, podendo ser editadas e lidas. Dessa forma, uma aplicação sendo analisada pelo Pin pode inspecionar quais as variáveis de ambiente estão criadas no momento de sua execução e quais os seus valores, permitindo a detecção da ferramenta DBI.

```
TLS Slots:
[0]:    0x0
[1]:    0x0
[2]:    0x0
[3]:    0x0
[4]:    0x0
[5]:    0x0
[6]:    0x0
[7]:    0x0
[8]:    0x0
[9]:    0x0
[10]:   0x0
[11]:   0x0
[12]:   0x0
[13]:   0x0
[14]:   0x0
[15]:   0x0
[16]:   0x0
[17]:   0x0
[18]:   0x0
[19]:   0x0
[20]:   0x0
```

```
TLS Slots:
[0]:    0x0
[1]:    0x90100
[2]:    0x16c0010
[3]:    0x0
[4]:    0x0
[5]:    0x0
[6]:    0x0
[7]:    0x0
[8]:    0x0
[9]:    0x0
[10]:   0x0
DBI Detected!!

DBI tool = PIN!
```

Figura 3.5: Comparação entre a estrutura TLS com e sem o Pin (Sun et al., 2016).

3.2.3 Detecção do Compilador JIT

Frameworks DBI possuem, como um dos principais componentes, o compilador *just-in-time* (JIT), que possui como entrada o executável da aplicação a ser analisada e como saída as instruções que de fato serão executadas. No entanto, como apontado por Polino et al. (2017), os compiladores JIT dos *frameworks* DBI geram muita atividade dentro do processo sendo analisado. Por exemplo, cada vez que o código gerado pelo compilador JIT precisa ser alocado em um Cache de Código, são necessárias chamadas a diversas APIs do sistema operacional para que seja feito o acesso à memória e a alocação de páginas de memória do processo sendo analisado. As subseções seguintes apresentam técnicas que detectam os efeitos das ações do compilador JIT no processo sob análise a fim de detectar as ferramentas DBI.

3.2.3.1 *DLL Hooks*

Quando o *framework* de instrumentação Pin está em execução, determinadas bibliotecas do sistema operacional são modificadas pelo *framework* DBI, que insere *hooks*, ou ganchos, em rotinas do sistema providas por essas bibliotecas. Assim, a ferramenta DBI pode alterar o comportamento dessas rotinas e das APIs que as utilizam.

Segundo Falcón and Riva (2012a), uma das bibliotecas alteradas pelo *framework* DBI é a *ntdll.dll*, responsável por exportar o conjunto de APIs nativas do sistema operacional (*for user-mode callers*) Eilam and Chikofsky (2005). De acordo com Rodriguez et al. (2016), o compilador JIT do framework Pin é o responsável por realizar essas alterações. Polino et al. (2017) acrescentam ainda que esses *hooks* podem ser encontrados sempre

no início das funções modificadas. Assim, uma aplicação sob análise de uma ferramenta DBI pode detectá-la através da inspeção das instruções iniciais de rotinas do sistema operacional que são normalmente modificadas por *frameworks* DBI.

3.2.3.2 Detecção Pelas Permissões de Páginas de Memória

Falcón and Riva (2012a) mostraram que a quantidade de páginas de memória com permissões de leitura, escrita e execução na memória virtual de um processo pode ser utilizada como critério para indicar a presença do *framework* Pin, uma vez que seu compilador JIT (*just-in-time*) faz uso intenso delas, criando-as, escrevendo-as e as executando.

Posteriormente, Rodriguez et al. (2016) e Polino et al. (2017) confirmaram a eficácia desta técnica em detectar o *framework* DBI Pin. Ademais, Sun et al. (2016) demonstraram que as permissões de páginas de memória podem ser utilizadas em conjunto com outra técnica de evasão, a *Code Cache Detection* para confirmar se um segmento de memória é um cache de código.

No entanto, Falcón and Riva (2012a) e Rodriguez et al. (2016) advertiram que utilizar a quantidade de páginas de memória com permissões de leitura, escrita e execução como critério para apontar a presença de ferramentas DBI pode levar a falsos positivos, uma vez que a própria aplicação sendo analisada pode possuir um compilador JIT ou alocar uma grande quantidade de páginas de memória com essas permissões.

3.2.3.3 Detecção por Chamadas Comuns a *APIs*

Para criar e gerenciar as páginas de memória com permissões de leitura, escrita e execução utilizadas durante o processo de instrumentação, o *framework* DBI Pin faz uso intenso de um conjunto de rotinas do sistema operacional (APIs), como a *ZwAllocateVirtualMemory* (Falcón and Riva, 2012a; Polino et al., 2017). Falcón and Riva (2012a) mostraram que é possível para uma aplicação sob análise do *framework* DBI Pin monitorar o uso dessas rotinas do sistema operacional, a fim de detectar a ferramenta DBI.

Polino et al. (2017) sugeriram que uma forma de uma aplicação cliente monitorar o número de chamadas a uma dessas rotinas é alterá-la, adicionando um contador de número de execuções. Assim, seria possível comparar o número de chamadas realizadas com um limiar de número de chamadas considerado normal.

Ademais, Falcón and Riva (2012a) advertiram que utilizar a intensidade de uso de determinadas APIs do sistema operacional, por exemplo *ZwAllocateVirtualMemory*, como critério para apontar a presença de *frameworks* DBI, pode levar a falsos positivos, uma vez que a própria aplicação sendo analisada pode possuir um compilador JIT ou alocar uma grande quantidade de páginas de memória através dessas rotinas.

3.2.4 Detecção de *Overhead*

É sabido que *frameworks* DBI inserem um custo computacional adicional na execução da aplicação cliente, observável como um *overhead* no tempo de execução (Nethercote, 2004; Luk et al., 2005b).

Falcón and Riva (2012a) mostraram que a diferença no tempo de execução entre uma aplicação sendo executada em modo nativo e sob instrumentação é tal que pode ser utilizada como critério para apontar a presença do *framework* Pin. Dessa forma, uma aplicação sob análise da ferramenta DBI poderia calcular o tempo decorrido para a execução de um trecho de código e comparar esse valor com um limiar pré-estabelecido associado a uma execução normal daquele trecho. Caso o tempo decorrido fosse maior que o limiar, isso seria um indício da presença do *framework* DBI. Além disso, a fim de evitar falsos positivos, Falcón and Riva (2012a) sugeriram que uma aplicação cliente poderia executar deliberadamente comportamentos que são mais custosos computacionalmente quando sob análise de um *framework* DBI, como o carregamento de bibliotecas, a fim de aumentar o *overhead* no tempo de execução.

Rodriguez et al. (2016) corroboraram com Falcón and Riva (2012a) quanto a eficácia da técnica para detectar o *framework* de instrumentação Pin, no entanto, alertam sobre a ocorrência de falsos positivos, uma vez que o tempo de execução de uma aplicação é muito dependente do ambiente. Polino et al. (2017) apresentaram diversas APIs do Windows que poderiam ser utilizadas para obter o tempo de execução decorrido de uma aplicação, como *NtQueryPerformanceCounter* e *GetTickCount*, e posteriormente revelar a presença do *framework* DBI.

3.3 Técnicas Indiretas de Evasão

A categoria Técnicas de Evasão Indiretas agrupa técnicas, também conhecidas como de Evasão Não Condicional *Non-Conditional Evasion* (Bulazel and Yener, 2017), que evadem *frameworks* de instrumentação binária dinâmica sem a necessidade de incorporar trechos de código para detectar a presença da ferramenta DBI. São técnicas que exploram limitações no funcionamento das ferramentas de análise, como instruções não suportadas, códigos que podem levar o instrumentador a exceções ou modos de execução do processador que não são suportados.

3.3.1 Limitação de Recursos

De modo geral, ambientes de análise, sejam manuais ou automáticos, não possuem recursos ilimitados. Dessa forma, características do ambiente (como número de processadores, núcleos e quantidade de memória disponível) e da análise (como o tempo disponível para realizar a observação e análise de uma amostra) são limitados. São essas limitações que são exploradas por técnicas de anti-instrumentação.

3.3.1.1 *Stalling Code*

Stalling Code é uma das técnicas que não visam detectar o *framework* de instrumentação binária dinâmica, mas sim postergar a execução dos comportamentos maliciosos, a fim de evadir as análises realizadas. Explora a limitação temporal da análise desempenhada pelos *frameworks* DBI, ou seja, o fato do tempo que um sistema pode gastar analisando uma única amostra ser limitado (Kolbitsch et al., 2011). Por isso, é uma técnica focada em ambientes automatizados de análise ou qualquer ambiente que impõe um limite de tempo de execução sobre a análise a ser realizada.

Kolbitsch et al. (2011) definem *stalling code* como uma sequência de instruções que atendem duas propriedades. A primeira diz que uma sequência de instruções que é executada consideravelmente mais lentamente no ambiente de análise do que em modo nativo. Ou seja, possui um conjunto de instruções que causam *overhead* no tempo de execução da aplicação quando em um ambiente de a análise. A segunda diz que o tempo total de execução do *stalling code* deve ser "*non-negligible*" em relação ao limite de tempo de análise imposto pelo ambiente de análise. De acordo com Kolbitsch et al. (2011), para que as duas propriedades sejam atendidas, *stalling code* é tipicamente implementado através de um *loop* que contém operações"lentas", ou seja, que levam um tempo considerável para serem executadas, como *system calls*.

Observando que *stalling code* é uma técnica de evasão que afeta as ferramentas de análise dinâmica de maneira geral, não especificamente os *frameworks* DBI, Rodriguez et al. (2016) propuseram seu uso para evadir o *framework* de instrumentação binária dinâmica Pin.

3.3.2 Limitação de Funcionalidade

Ambientes de análise comumente precisam ser capazes de reproduzir comportamentos padrões de ambientes *bare metal* (ambientes nativos), do sistema operacional, entre outros, para serem capazes de realizar as análises sobre as aplicações cliente. No entanto, nem sempre todos os comportamentos de um ambiente são tratados e implementados nas ferramentas de análise, levando a inconsistências comportamentais entre um ambiente de análise e um ambiente nativo, ocasionando falhas de execução. Essas diferenças podem levar uma aplicação cliente a evadir a análise de uma ferramenta DBI.

3.3.3 Instruções Não Suportadas

Conforme mostrado por Sun et al. (2016), nem todas as instruções do conjunto de instruções de uma arquitetura tem execução suportada pelos *frameworks* de instrumentação binária dinâmica. Para o *framework* DBI Pin, por exemplo, a execução da instrução *assembly "far ret"*, por uma aplicação sob análise, resulta no encerramento da aplicação e do *framework* de instrumentação com a mensagem: "*Pin doens't support FAR RET [...]*", tornando a ferramenta DBI incapaz de analisar a aplicação. Em muitos casos, a própria documentação do *framework* DBI não lista o conjunto de instruções não suportadas.

3.3.4 *Crash Code*/Comportamentos Não Suportados

Ligh et al. (2014) foram os primeiros a sugerir, como técnica de evasão, o uso de artefatos de código que são corretamente executados quando a aplicação está sendo executada em modo nativo, mas levam a exceções quando sob análise de um *framework* de instrumentação binária dinâmica. No caso dos autores, o DynamoRIO.

Posteriormente, Sun et al. (2016) mostraram que o uso de funcionalidades do sistema operacional, como *WoW64* que permite a execução de códigos de arquiteturas 32 bits e 64 bits na mesma aplicação em sistemas operacionais Windows, podem levar o *framework* DBI DynamoRIO a exceções e, consequentemente, ao encerramento abrupto.

Apesar das semelhanças, Instruções Não Suportadas e *Crash Code*/ Comportamentos Não Suportados são técnicas de evasão diferentes. No primeiro caso, o *framework* de instrumentação encerra a aplicação sob análise e então realiza uma saída graciosa (*graceful exit*), devido a uma incompatibilidade com uma instrução não suportada, gerando uma notificação disso para o analista da ferramenta DBI. No segundo caso, ocorre o encerramento abrupto da aplicação e do *framework* de instrumentação binária dinâmica devido a uma exceção não tratada corretamente.

3.4 Discussão

A Tabela 3.1 sumariza as técnicas de evasão de *frameworks* DBI apresentadas, os trabalhos que as propuseram e quais dessas técnicas são atualmente funcionais contra as versões mais recentes dos *frameworks* alvejados.

A coluna **Técnicas de Evasão** lista as técnicas anti-instrumentação analisadas. A coluna **Trabalhos** enumera os autores (artigos) que propuseram as técnicas de evasão. A coluna **PoC** informa se a técnica evasiva é funcional contra o *framework* DBI alvejado. Dessa forma, um traço ("-") indica que não existe prova de conceito disponibilizada pelo autor. Por fim, a coluna Fuga DBI explicita as técnicas anti-instrumentação que podem levar a fuga da instrumentação binária dinâmica, através da exploração da superfície de ataque inseridas pelas ferramentas DBI.

É importante destacar que a eficácia das técnicas de evasão foi verificada através da execução das provas de conceito disponibilizadas pelos autores dos trabalhos, sobre a versão mais recente dos *frameworks* DBI alvejados por elas. Infelizmente, as provas de conceito disponíveis são apenas focadas no *framework* DBI Pin. Assim, elas foram testados com a versão mais recente do *Framework* Pin (versão 3.6.97554), em um equipamento com Processador Intel Core i5-3230M 2.60 GHz, 8GB de memória RAM, sistema operacional Windows 7 Ultimate x64, compiladores Microsoft C/C++ Optimizing Compiler Version 15.00.30729.01 para x64, Microsoft C/C++ Optimizing Compiler Version 16.00.30319.01 para 80x86 com Microsoft Visual Studio 2010. Por fim, uma vez que o sistema operacional Windows 7 foi o sistema alvejado pelas provas de conceito propostas na literatura, o mesmo foi adotado como sistema operacional do ambiente de testes.

Das 33 técnicas apresentadas, apenas 9 delas possuem provas de conceito (PoC).

Tabela 3.1: Relação de Técnicas de Evasão por Trabalho

Técnicas de Evasão	**Trabalhos**								**PoC**	**Fuga**	**Classificação**
Detecção de EIP/RIP			X	X		X	X	X	Sim	Não	Artefatos de Cache de Código
Código Automodificante			X				X	X	-	Não	Artefatos de Cache de Código
Estados da Cache de Código vs Memória Virtual			X						Sim	Não	Artefatos de Cache de Código
Fraquezas Inerentes à Proteção de Escrita Quando Ocorre Código Automodificante			X						Não	Não	Artefatos de Cache de Código
Contexto Inesperado	X								-	Sim	Artefatos de Cache de Código
Exceções Inesperadas		X							-	Não	Artefatos de Cache de Código
Xmode Code	X								-	Não	Artefatos de Cache de Código
Abuso das Instruções de *syscall*								X	-	Não	Artefatos de Cache de Código
Emulação incorreta da instrução *rdfsbase*								X	-	Não	Artefatos de Cache de Código
Negligenciamento do *No-eXecute Bit* (nx)								X	Sim	Sim	Artefatos de Cache de Código
Detecção do Processo Pai		X		X		X	X		Sim	Não	Artefatos de Ambiente
Memory Fingerprinting / Detecção de Cache de Código	X						X		Sim	Sim	Artefatos de Ambiente
Detecção Por Argv				X		X			Não	Não	Artefatos de Ambiente
Handle de Processo				X		X			-	Não	Artefatos de Ambiente
Número de *File Handlers*		X							-	Não	Artefatos de Ambiente
Número Máximo de *File Handlers* Abertos		X							-	Não	Artefatos de Ambiente
Pico no Uso de Memória		X							-	Não	Artefatos de Ambiente
Alteração no Número Máximo de *File Handlers*		X							-	Não	Artefatos de Ambiente
Detecção pela Máscara de Sinal (*Signal Masks*)		X							-	Não	Artefatos de Ambiente
Fingerprinting pinvm.dll/Detecção de Bibliotecas do DynamoRIO		X		X		X			Não	Não	Artefatos de Ambiente
Detecção Pelas Funções Exportadas por *Pintools*				X		X			Não	Não	Artefatos de Ambiente
Detecção Pelos Nomes de Seções				X		X			Não	Não	Artefatos deAmbiente
Detecção Por *Event Handles*				X		X			Não	Não	Artefatos de Ambiente
Detecção Por *Section Handles*				X		X			Não	Não	Artefatos de Ambiente
Detecção por *Thread Local Storage* (TLS)	X								Sim	Sim	Artefatos de Ambiente

[1] (Sun et al., 2016); [2] (Ligh et al., 2014); [3] (Hron and Jermář, 2014); [4] (Falcón and Riva, 2012a); [5] (Kolbitsch et al., 2011); [6] (Rodriguez et al., 2016); [7] (Polino et al., 2017); [8] (Zhechev, 2018).

Tabela 3.2: Continuação da Relação de Técnicas de Evasão por Trabalho

Técnicas de Evasão	Trabalhos								PoC	Fuga	Classificação
	1	2	3	4	5	6	7	8			
Variáveis de Ambiente Internas do Pin								X	-	Não	Artefatos de Ambiente
DLL Hooks				X		X	X		Sim	Sim	Detecção do Compilador JIT
Detecção Pelas Permissões de Páginas de Memória	X			X		X	X	X	Sim	Não	Detecção do Compilador JIT
Detecção por Chamadas Comuns a *APIs*				X			X		Não	Não	Detecção do Compilador JIT
Detecção de Overhead		X		X		X	X	X	Sim	Não	Detecção de Overhead
Stalling Code					X				-	Não	Limitação de Recursos
Instruções Não Suportadas	X								-	Não	Limitação Funcional
Crash Code/Comportamentos Não Suportados	X	X							-	Não	Limitação Funcional

[1] (Sun et al., 2016); [2] (Ligh et al., 2014); [3] (Hron and Jermář, 2014); [4] (Falcón and Riva, 2012a); [5] (Kolbitsch et al., 2011); [6] (Rodriguez et al., 2016); [7] (Polino et al., 2017); [8] (Zhechev, 2018).

Mais especificamente, apenas dois autores de fato apresentam provas funcionais. Falcón and Riva (2012a), através da ferramenta *eXait*, apresenta PoCs para *Detecção de EIP*, *Detecção do Processo Pai*, *DLL Hooks*,*Detecção Pelas Permissões de Páginas de Memória* e *Detecção de overhead*. Hron and Jermář (2014) propôs PoCs para as técnicas de evasão *Detecção de EIP*, *Estados da Cache de Código vs Memória Virtual* e *Fraquezas Inerentes à Proteção de Escrita Quando Ocorre Código Automodificante*.

Um detalhe que é claramente percebido na Tabela é quais são as técnicas que de fato promovem a fuga do processo de instrumentação binária dinâmica, ou seja, que conseguem de forma mais imediata aumentar à superfície de ataque explorando vulnerabilidades das ferramentas DBI. Em linhas gerais, essas técnicas exploram as estruturas dos *frameworks* DBI para realizar fuga, como previamente discutido na seção 2.3.

As técnicas de ***Memory Fingerprinting*/Detecção de Cache de Código**, **Detecção por *Thread Local Storage* (TLS)** e *Contexto Inesperado* exploram os Caches de Código como alvo essencial para concretização da fuga de instrumentação em técnicas como a *Run, Modify, Run*, oferecendo, claramente, o risco de que a aplicação cliente realize a fuga da instrumentação, caso não sejam mitigadas. Essas técnicas corroboram com as ideias apresentadas por Sun et al. (2016).

Já as técnicas de evasão *Contexto Inesperado*, *DLL Hooks* e **Negligenciamento do *No-eXecute Bit* (nx)** interagem com outras estruturas críticas das ferramentas DBI, que não Caches de Código - excessão a técnica *Contexto Inesperado* -, para realizar a fuga da instrumentação. A técnica *Contexto Inesperado*, por exemplo, permite a interação entre a aplicação cliente e o contexto da ferramenta DBI, através do registrador

EBX. Já a técnica *DLL Hooks* expõe as instruções de controle de fluxo de execução inseridas pelos *frameworks* DBI nas bibliotecas do sistema operacional. Por fim, a técnica **Negligenciamento do *No-eXecute Bit* (nx)** explora uma vulnerabilidade do *framework* DBI, uma vez que negligencia um importante mecanismo de segurança dos sistemas operacionais (e processadores), permitindo a execução de dados como código. Contudo, embora isso torne possível a concretização de uma gama de comportamentos maliciosos, isso não leva a fuga da instrumentador.

É importante destacar que as técnicas que promovem fuga da instrumentação ou exploram vulnerabilidades do *framework* descritas em negrito são as que esta dissertação irá apresentar novas contramedidas no Capítulo 5.

Capítulo 4

Mitigação a Evasão de Frameworks DBI

O capítulo anterior sumarizou as técnicas evasivas anti-instrumentação apresentadas na literatura. Este Capítulo apresenta os mecanismos existentes para evitar a concretização das técnicas de evasão de *frameworks* DBI. Nem todas as técnicas de evasão apresentadas anteriormente são mitigadas atualmente. Grande parte de tal fato se deve aos mecanismos de mitigação propostos geralmente serem focados em um *framework* DBI específico, podendo não funcionar como esperado em outras ferramentas DBI.

4.1 Detecção de EIP/RIP

Quando uma aplicação está sendo analisada por um *framework* DBI, o código em execução da aplicação é proveniente de regiões de memória conhecidas como Caches de Código, ao invés da região de memória onde o executável foi carregado inicialmente. Essa diferença na região de memória dos endereços das instruções sendo executadas pode ser utilizada para detectar a ferramenta de instrumentação binária dinâmica. Uma forma de obter o endereço da próxima instrução a ser executada é analisando o conteúdo do registrador *Instruction Pointer* (EIP ou RIP), o que pode ser feito através do uso de instruções especiais que salvam em memória o contexto de execução do programa em execução, o que inclui o conteúdo do registrador EIP.

Polino et al. (2017) propuseram como contramedida a essa técnica de evasão monitorar as chamadas às instruções que carregam em memória o valor do registrador EIP, como *FSTENV*, *FSAVE*, *FXSAVE* e *int 2e*, e quando realizada a chamada a alguma dessas instruções, alterar seu o retorno, a fim de que o verdadeiro valor do registrador EIP não seja acedido. Para isso, na solução `Arancino` todas as instruções executadas são inspecionadas. Quando uma das instruções citadas é detectada, é adicionado imediatamente após ela, uma sequência de códigos para alterar o valor do EIP retornado, fazendo-o apontar para um endereço de memória de uma instrução proveniente da região onde o executável foi carrregado originalmente.

4.2 Código Automodificante

Considerando que durante a análise de uma aplicação por um *framework* DBI o código original da aplicação não é executado, mas sim uma versão modificada que reside em Caches de Código, os *frameworks* DBI precisam ser capazes de lidar com códigos automodificantes (*self-modifying* ou SMC), que, como o próprio nome diz, alteram instruções do programa sendo executado, em tempo de execução. Caso os códigos automodificantes não sejam tratados corretamente, podem levar a aplicação cliente ao encerramento abrupto ou a revelar a presença do ambiente análise DBI.

Polino et al. (2017) propuseram, como contramedida para essa técnica de evasão, detectar as alterações realizadas por código automodifcante nas instruções da aplicação em execução e em seguida forçar o *framework* DBI a reconstruir a Cache de Código com o código modificado. O momento da modificação do código da aplicação em execução pode ser determinado através da monitoração das instruções de escrita em memória. Tentativas de escrita em regiões de memória onde o código da aplicação está carregado caracterizam a ação de SMC.

Apesar dessa solução ser eficaz contra várias implementações da técnica de evasão por *Self-Modifying Code*, ela é muito similar a contramedida nativa empregada pelo *framework* Pin, descrita por Zhechev (2018). Um estudo comparativo da eficácia e dos custos computacionais de cada uma seria apropriado.

4.3 Detecção do Processo Pai

Segundo Polino et al. (2017), para mitigar a técnica de detecção do *framework* DBI através do processo pai, é necessário monitorar as chamadas aos mecanismos que podem ser utilizados por uma aplicação sob instrumentação para obter informações sobre os processos em execução no sistema operacional ou sobre sua própria árvore de processos. Segundos os autores, existem duas soluções. A primeira é utilizar a API do sistema operacional *NtQuerySystemInformation* para obter informações sobre cada um dos processos em execução no sistema operacional (Microsoft, 2018f). Na contramedida apresentada, as chamadas a API *NtQuerySystemInformation* são interceptadas e tem seu retorno modificado obedecendo a seguinte regra: dados textuais contendo os caracteres "pin.exe"são substituídos por "cmd.exe". Dessa forma, referências ao nome do processo do *framework* DBI Pin são alteradas para referenciar o Terminal de Comandos do SO (*Windows Command Prompt*).

A segunda solução é relacionado ao processo do sistema operacional *CSRSS.exe*, um serviço crítico do SO que mantém sua própria lista dos processos em execução (Microsoft, 2018a). A contramedida é elaborar, em tempo de execução, uma lista de processos que não devem ser acessíveis pela aplicação cliente (*black list*), o que inclui o processo *CSRSS.exe*. Em seguida, a API do sistema operacional *NtOpenProcess* (Microsoft, 2018e), que permite a interação entre processos, é monitorada a fim de impedir a comunicação com os processos presentes nessa lista. Quando a API *NtOpenProcess* é invocada para realizar uma interação com algum dos processos presentes na lista, o

retorno da chamada de sistema é modificado para ser a constante *NTSTATUS STATUS_ACCESS_DENIED*, o que significa que o processo que fez a chamada à API não tem direitos de acesso ao objeto solicitado.

4.4 *Memory Fingerprinting*/Detecção de Cache de Código

Para mitigar a técnica de evasão *Memory Fingerprinting* (ou Detecção de Cache de Código), Polino et al. (2017) propuseram que determinadas APIs do sistema operacional que permitem realizar interações, como consultas e acessos a regiões de memória virtual, sejam monitoradas a fim de evitar consultas por artefatos de memória relacionados aos *frameworks* DBI.

O mecanismo de mitigação proposto pelos autores intercepeta e controla os resultados das chamadas às APIs do Windows *VirtualQuery* e *NtQueryVirtualMemory*, que retornam informações sobre um conjunto de páginas de memória do espaço de endereçamento virtual do processo (Microsoft, 2018j). Na solução apresentada, o endereço de memória passado para essas funções é verificado e a ferramenta de proteção do *framework* DBI avalia se o endereço pode ser acessado pela aplicação cliente.

Para determinar quais endereços podem ser acessados pela aplicação cliente, ou não (como os Caches de Código), podem ser utilizadas políticas de *whitelist* - listas de endereços que podem ser apresentados ou acessados pela aplicação cliente de acordo com regras preestabelecidas (Computer Security Resource Center, 2018) - e *blacklist* - inverso da *whitelist*- nos segmentos de memória do processo.

Polino et al. (2017) adotaram uma política de *whitelist*, onde uma lista de endereços que podem ser acessados pela aplicação cliente é criada no início do processo de instrumentação e atualizada durante a execução. Dessa forma, toda vez que uma das APIs do Windows *VirtualQuery* ou *NtQueryVirtualMemory* é chamada, é verificado se o endereço passado está na *whitelist*. Em caso positivo, a execução da API se dá normalmente. Em caso negativo, o retorno da chamada a API é alterado para indicar que o endereço consultado possui o status *MEM_FREE* (Microsoft, 2017b) que indica que a página de memória a qual o endereço pertence está livre (vazia), mas não pode ser acessada, nem alocada pelo processo que está fazendo a requisição.

4.5 Detecção pelas Funções Exportadas por *Pintools* e Detecção pelos Nomes de Seções

Segundo Rodriguez et al. (2016), as técnicas de evasão Detecção Pelas Funções Exportadas por *Pintools* e Detection Pelos Nomes de Seções são baseadas no uso de determinadas APIs do sistema operacional que retornam dados textuais referentes ao *framework* de instrumentação, como o nome de funções exportadas pelas *pintools* ou nomes de seções referentes ao Pin. Por isso, a contramedida proposta pelos autores é baseada no monitoramento das chamadas a funções do sistema operacional, seguida

pela verificação se algum dos seu retornos contém informação textual relacionada ao Pin. Em caso positivo, a informação textual é alterada para não revelar a presença da ferramenta DBI. No entanto, os autores não entraram em detalhes sobre quais APIs do sistema operacional são monitoradas atualmente e como se dá a modificação da informação textual original.

4.6 *DLL Hooks*

Frameworks de instrumentação alteram algumas DLLs (*dynamic-link library*) do sistema operacional, como a *ntdll.dll*, inserindo *hooks*, ou ganchos, que desviam o fluxo de execução normal da biblioteca para o *framework* de instrumentação. Essas alterações podem ser detectadas pela aplicação cliente por meio de uma verificação no código dessas DLLs. Além disso, esses ganchos podem ser removidos caso a aplicação cliente reescreva o código carregado em memória dessas bibliotecas.

Para impedir que a aplicação cliente detecte as modificações realizadas pelas ferramentas DBI nas bibliotecas do SO, Polino et al. (2017) propuseram que ao invés de permitir que a aplicação consulte as bibliotecas modificadas pelo *framework* de instrumentação, que essa consulta seja redirecionada para uma versão não modificada dessas bibliotecas, garantindo a transparência dos *hooks* colocados pelo *framework* de instrumentação. No entanto, chamadas a APIs do sistema ainda são direcionadas para as versões modificadas pelo *framework* de instrumentação.

A implementação proposta por Polino et al. (2017) consiste em inicialmente gerar uma lista de bibliotecas do sistema que precisam ser protegidas e então, para realizar o redirecionamento das consultas às bibliotecas dessa lista, todas as instruções de leitura de memória virtual são monitoradas. Assim, quando é detectado que aplicação cliente está tentando ler um endereço pertencente a uma das bibliotecas modificadas pelo *framework* DBI, a instrução de leitura é modificada para apontar para a versão sem *hooks* da biblioteca.

4.7 Detecção pelas Permissões de Páginas de Memória

Como apresentado na seção 2.3, a estrutura de um processo sob a análise de um *framework* DBI é diferente de quando está em modo nativo, sem instrumentação. No primeiro caso, passam a existir mais segmentos de código, conhecidos como Caches de Código que têm como principal característica possuírem permissões de escrita, leitura e execução.

Assim, a quantidade de seções de memória com permissões de leitura, escrita e execução pode ser utilizada como critério para determinar a presença de Caches de Código e consequentemente de *framework* de instrumentação binária dinâmica. APIs do sistema operacional, como *VirtualQuery*, podem ser utilizadas para varrer o espaço de memória do processo, a fim de contabilizar os segmentos de memória com permissões de leitura, escrita e execução. Além disso, uma segunda maneira de verificar as permissões

de um endereço de memória é por meio de acesso direto ao endereço de memória, ou seja, através de uma tentativa de leitura, escrita e execução.

O mecanismo de mitigação para esta técnica de evasão, desenvolvido por Polino et al. (2017), é semelhante ao adotado contra a técnica de evasão *Memory Fingerprinting*/Detecção de Cache de Código. Inicialmente é construída uma lista de endereços que podem ser acessados pela aplicação cliente, *whitelist*. Em seguida, as APIs do sistema operacional que permitem consulta e acesso a páginas de memória são monitoradas a fim de evitar consultas a regiões de memória que não constam na *whitelist*.

4.8 Detecção por Chamadas Comuns a APIs

Uma aplicação cliente pode detectar a presença do *framework* de instrumentação binária dinâmica alterando as APIs do sistema operacional que são intensamente utilizadas pelas ferramentas DBI. Pode-se, por exemplo, adicionar um contador de número de chamadas em uma dessas APIs, com o objetivo de detectar o uso intenso das mesmas.

A contramedida para esta técnica, proposta por Polino et al. (2017), é a monitoração das operações de escritas realizadas sobre as APIs comumente utilizadas pelo *framework* de instrumentação. Caso detectada uma tentativa de escrita a uma dessas APIs, ocorre o redirecionamento da escrita e das tentativas de leitura àquela região. Assim, evita-se que seja feito o processo de contagem de chamadas do sistema operacional e inserção de demais modificações nas APIs utilizadas pelo *framework* DBI. Essa contramedida pode ser implementada através do uso do nível de granularidade por instrução.

4.9 Detecção de *Overhead*

Segundo Polino et al. (2017), existem casos onde, para mitigar as técnicas de evasão que utilizam medidas de *Overhead* de tempo, não é suficiente monitorar as APIs do sistema operacional que retornam o tempo do sistema, como *GetTickCount*, *timeGetTime*. Isso porque é possível obter as informações retornadas por essas APIs sem chamá-las, através de uma consulta a página de memória compartilhada KUSER_SHARED_DATA (Microsoft, 2017a).

Assim, Polino et al. (2017) adotaram como contramedida interceptar tentativas de leituras a região de memória onde reside a página KUSER_SHARED_DATA e alterar as informações lidas, de forma a garantir que não revelem a presença de *overhead*. De forma prática, o nível de granularidade por instrução pode ser utilizado para monitorar as instruções que fazem leituras na memória e modificar os tempos retornados por elas, reduzindo-os.

Outra maneira de obter um referencial de tempo de execução da aplicação é utilizando a instrução *rdtsc*, que retorna o *timestamp* do processador nos registradores *EDX* e *EAX*. A abordagem da contramedida nesse caso é a utilização do nível de granularidade por instrução para monitorar a chamada desta instrução e alterar o seu retorno nos registradores.

Apesar da solução proposta por Polino et al. (2017) ser capaz de mitigar diversas implementações da técnica de Detecção de *Overhead*, não é capaz impedir a efetivação de todas. Segundo Qiu et al. (2014), técnicas de mitigação baseadas em monitorar e alterar o retorno de instruções e de chamadas a APIs do sistema operacional não são capazes de conter implementações que utilizam informações temporais de fontes externas, como de rede, *network time server* e internet.

4.10 Stalling Code

A evasão de *frameworks* DBI pelo uso de *stalling code* consiste em postergar a execução do código cliente através do uso de instruções para adormecer o processo por determinado tempo e/ou de códigos inseridos com o objetivo de serem executados repetidamente. Essa técnica de evasão possui duas características, segundo (Kolbitsch et al., 2011): o *stalling code* foca inicialmente em utilizar instruções que causam maior atraso temporal quando a aplicação está sob análise da ferramenta DBI do que quando está executando nativamente; e a duração da execução da aplicação deve ser "não negligenciável" em relação ao limite de tempo imposto pelo ambiente de análise.

Baseadas nas características da técnica de *stalling code*, diferentes contramedidas foram propostas por Kolbitsch et al. (2011). A primeira delas é aumentar, ou remover quando possível, o limite de tempo imposto pelo ambiente de análise sobre a aplicação. Dessa forma, mesmo com a ocorrência de instruções com o objetivo de atrasar a execução do código da aplicação, seria possível analisar a aplicação cliente. Outra contramedida seria monitorar o uso e a frequência das instruções que causam maior atraso temporal sobre o ambiente de análise, como chamadas de sistema. Quando essas instruções fossem detectadas consecutivamente, bastaria não realizar análises DBI sobre elas. Uma terceira contramedida que poderia ser adotada é realizar uma análise de progresso no código sendo executado, em tempo de execução, para verificar se o mesmo trecho está sendo executado repetidamente. Caso fosse confirmado que o código não progride, bastaria desativar a análise DBI sobre o código sendo executado múltiplas vezes. Vale ressaltar que as contramedidas apresentadas são complementares, nem sempre sendo possível utilizá-las simultaneamente, segundo os autores.

4.11 Discussão

Dentre as 33 técnicas apresentadas, foram propostas contramedidas na literatura para 12. Com isso e considerando o aumento da popularidade dos *frameworks* de instrumentação binária dinâmica, espera-se que as técnicas anti-instrumentação continuem atraindo a atenção de pesquisadores em busca de soluções contra a evasão das ferramentas DBI e dos desenvolvedores de *malwares*, mantendo o *arms race* na área. A Tabela 4.1 enumera as técnicas anti-instrumentação que possuem contramedidas propostas pela literatura e os trabalhos que propuseram as contramedidas. Comparando as Tabelas 3.1 e 4.1 é possível notar que um número diminuto de técnicas de evasão é mitigado. Existem casos também, como o da técnica *Memory Fingerprinting*/Detecção de Cache de Código, em

que nem todas as variações da mesma são tratados. Especificamente para esse caso, não há contramedida quando são buscados os Caches de Código através do padrão hexadecimal inicial dos Caches de Código (*0xFEEDBEAF*). Dessa forma, ambientes de análise que empregam tais ferramentas estão expostos a fuga da instrumentação binária dinâmica. Devido a isso, Zhechev (2018) considerou que o *frameworks* de instrumentação binária dinâmica eram ferramentas inapropriadas para a análise de software não seguro, ou potencialmente malicioso, como *malwares*.

Tabela 4.1: Relação de Técnicas de Evasão com Contramedidas

Técnicas de Evasão	**Trabalhos**			**Classificação**
	1	2	3	
Detecção de EIP			X	A. de Cache de Código
Código Automodificante			X	A. de Cache de Código
Contexto Inesperado			X*	A. de Cache de Código
Detecção do Processo Pai			X	A. de Ambiente
Memory Fingerprinting / Detecção de Cache de Código			X*	A. de Ambiente
Detecção Pelas Funções Exportadas por *Pintools*		X		A. de Ambiente
Detecção Pelos Nomes de Seções		X		A. de Ambiente
DLL Hooks			X	D. do Compilador JIT
Detecção Pelas Permissões de Páginas de Memória			X	D. do Compilador JIT
Detecção por Chamadas Comuns a APIs			X	D. do Compilador JIT
Detecção de *Overhead*			X	Detecção de Overhead
Stalling Code	X			Limitação de Recursos

[1] (Kolbitsch et al., 2011).
[2] (Rodriguez et al., 2016).
[3] (Polino et al., 2017).
[*] Não há contramedidas para todas as variações da técnica de evasão.

As técnicas de mitigação que propõem monitoração de APIs do sistema operacional possuem o lado negativo de necessitarem que todas as APIs que fornecem determinada

funcionalidade alvo sejam monitoradas. Além disso, como alteram o retorno de uma chamada ao sistema operacional, feita pela aplicação cliente, podem haver condições de *arms race* - a aplicação cliente detectar o algoritmo de mitigação da técnica evasiva e determinar que está sob análise por meio de retornos diferentes do esperado às chamadas de sistema operacional. As técnicas de mitigação que utilizam análises de granularidade de instrução, apesar de fornecerem controle total sobre a aplicação sob análise e sofrerem menos com condições de *arms race*, tem a desvantagem de elevar o custo computacional consideravelmente em relação aos outros níveis de granularidade.

Além disso, este é o primeiro trabalho que faz um levantamento extensivo, considerando apresentações em conferências, das técnicas de evasão e de mitigação existentes e organizá-las de forma compreensível através de uma taxonomia. Trabalhos anteriores como de Sun et al. (2016) e Polino et al. (2017) propuseram novas técnicas, tanto de evasão, quanto de mitigação e preocuparam-se em propor taxonomias exclusivamente para as técnicas abordadas por eles.

O Capítulo seguinte apresenta novas contramedidas para mitigar as técnicas de evasão que podem levar a fuga da instrumentação (que hoje não possuem contramedidas funcionais).

Capítulo 5

Novas Contramedidas

Este Capítulo apresenta as contramedidas propostas para algumas técnicas evasivas ainda sem solução, discutindo as implementações das provas de conceito (PoC) desenvolvidas e os impactos no desempenho e eficácia ocasionados por seu uso (*overhead* temporal). Primeiramente (seção 5.1) é apresentada a ferramenta `PinVMShield`, derivada do *framework* Pin, empregada na implementação das contramedidas. Na seção 5.2, cada uma das novas contramedidas é apresentada com sua respectiva prova de conceito (PoC). Após isso, na seção 5.3, a eficácia das contramedidas propostas é verificada através de testes com provas de conceito das técnicas anti-instrumentação e os impactos de desempenho associados (*overhead* temporal) são medidos através do *benchamrk* SPEC. Por fim, é feita uma análise se os *frameworks* DBI, especificamente o Pin, tornam-se ferramentas mais seguras (com menor superfície de ataque explorável) quando as técnicas evasivas que podem levar a fuga da instrumentação são mitigadas.

5.1 *PinVMShield*

`PinVMShield` (Rodriguez et al., 2016) é uma ferramenta baseada no *framework* DBI Pin para detectar e mitigar técnicas de evasão utilizadas por *malwares* cientes de análise (*analysis-aware malware*). A ferramenta possui uma estrutura de *plugins*, mostrada na Figura 5.1, que permite a extensão das funcionalidades presentes na ferramenta. Assim, é possível extendê-la para cobrir diferentes técnicas evasivas. É distribuída sob a licença GNU GPL versão 3 e possui o código fonte disponível em `https://bitbucket.org/rjrodriguez/pinvmshield/`.

`PinVMShield` foi desenvolvida para sistemas operacionais Windows, apesar de poder ser estendida para outros sistemas operacionais que o Pin tem suporte. Possui dois níveis de granularidade para análise: nível de rotina e nível de instrução. Esses dois níveis de granularidade permitem que a ferramenta detecte comportamentos evasivos baseados em APIs do Windows que são utilizados contra *debuggers*, como por exemplo `IsDebuggerPresent` ou `CheckRemoteDebugger`.

Atualmente, a ferramenta foca em comportamentos evasivos contra ambientes vir-

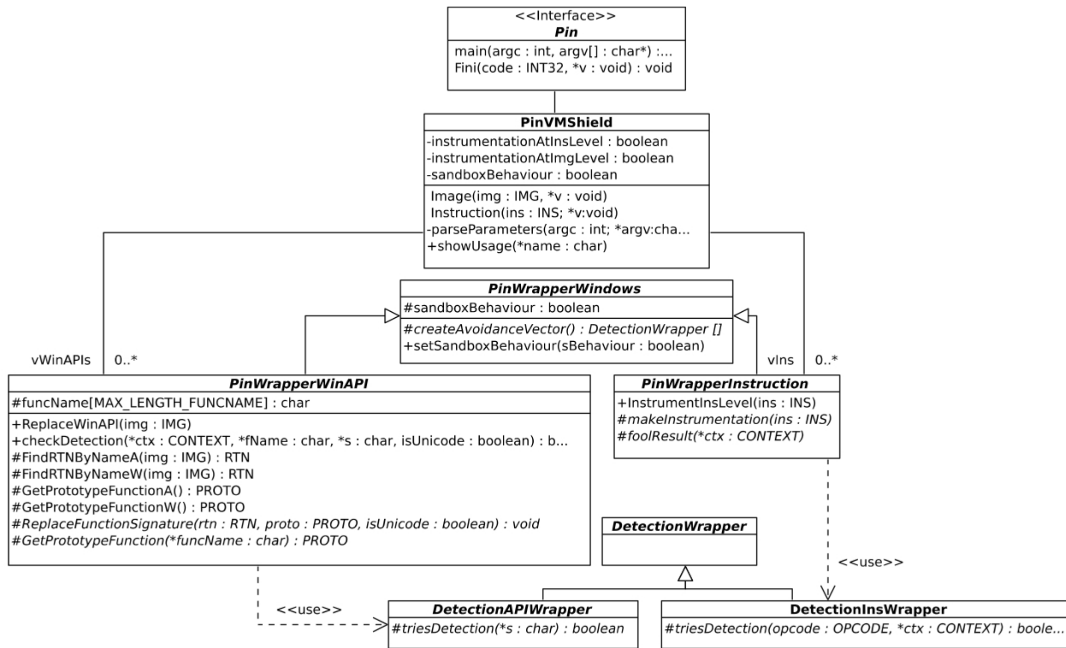

Figura 5.1: Diagrama de classes do `PinVMShield` (Rodriguez et al., 2016)

tualizados (em particular, *VirtualPC, VMWare, e Virtualbox*), *debuggers* (*WinDBG, OllyDBG, e ImmunityDebugger*) e *sandboxes* (*WinJail, Cuckoo Sandbox, Norman, Sandboxie, CWSandbox, JoeSandbox, e Anubis*).

As contramedidas apresentadas nesta dissertação foram desenvolvidas como *plugins* do PinVMShield devido a facilidade no desenvolvimento das soluções agregada pelo uso dessa ferramenta. Especificamente, os protótipos desenvolvidos, discutidos em detalhes nas seções subsequentes 5.2.1, 5.2.2 e 5.2.3, são classes que herdam os atributos e métodos da classe abstrata `PinWrapperWinAPI`, visível na Figura 5.1, o que permite que seja feito com facilidade a interceptação de chamadas a APIs do Windows, através do método `ReplaceWinAPI`. Assim, os protótipos desenvolvidos utilizam apenas a granularidade a nível de rotina.

5.2 Contramedidas

Esta seção descreve as novas soluções de mitigação para técnicas anti-instrumentação que podem levar a fuga da instrumentação e que não possuem contramedidas eficazes na literatura. Os mecanismos de mitigação propostos são baseados em proposições feitas pelos autores das técnicas evasivas quanto ao funcionamento e modo de operação de cada técnica. São discutidos detalhes implementacionais que norteiam as decisões de projeto que levaram a implementação das provas de conceito das contramedidas apresentadas anteriormente. São apresentadas as implementação de cada uma das técnicas de evasão

mitigadas, Detecção por *Thread Local Storage* (TLS), *Memory Fingerprinting* / Detecção de Cache de Código, Negligenciamento do *No-eXecute Bit* (nx). Vale destacar que as provas de conceito desenvolvidas estão disponíveis online[1], além de encontrarem-se listadas no apêndice B.

5.2.1 Detecção por *Thread Local Storage* (TLS)

Esta subseção apresenta a contramedida proposta contra a técnica anti-instrumentação Detecção por *Thread Local Storage* (TLS), iniciando por uma definição do que é *Thread Local Storage* e como a técnica de detecção de TLS funciona. Por fim é introduzida a contramedida proposta.

5.2.1.1 *Thread Local Storage* (TLS)

Como introduzido na seção 3.2.2.15, *Thread Local Storage*, ou simplesmente TLS, é uma funcionalidade que permite que threads, que têm suas variáveis estáticas e globais naturalmente compartilhadas entre todas as threads do mesmo processo, utilizem variáveis globais (índice do vetor TLS) com conteúdos privados para cada thread. De maneira direta, as posições do TLS podem ser vistas como variáveis globais que possuem conteúdo particular, manipulado individualmente por cada thread do processo.

A Figura 5.2 ilustra duas *threads* de um mesmo processo utilizando a estrutura de dados fornecida pelo TLS para armazenar dados de maneira privada a cada thread.

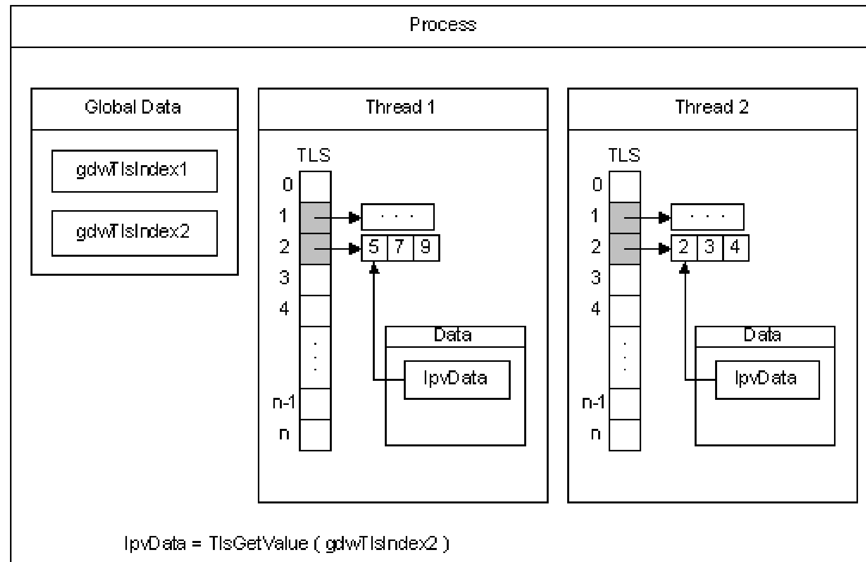

Figura 5.2: Ilustração da TLS sendo utilizada por duas threads de um processo (Microsoft, 2018i).

[1] See https://github.com/ailton07/PinVMShield

Como mencionado no Capítulo 2, os *Frameworks* DBI fazem uso intensivo de *Thread Local Storage* (TLS) para controlar os endereços Caches de Código críticos para a execução da ferramenta DBI e não tomam nenhuma medida para proteger essa estrutura. Segundo o manual da Microsoft (2018i), no Windows toda a interação com o TLS é intermediada pelo sistema operacional através das APIs: *TlsAlloc*, *TlsSetValue*, *TlsGetValue*, *TlsFree*. Assim, como contramedida para essa técnica anti-instrumentação é mandatório estabelecer o controle das chamadas a essas APIs do sistema operacional e determinar uma política de quais posições da estrutura de dados, em formato de vetor, do TLS será acessível pela aplicação cliente.

5.2.1.2 *Contramedida*

A política de controle de posições do TLS consiste em deslocar a interação entre a aplicação cliente e a estrutura de dados do TLS, através de um algoritmo de deslocamento circular simples em `N`, onde `N` é o número de posições do TLS sendo utilizadas pela ferramenta DBI. Dessa forma, se o *framework* de instrumentação utiliza as posições 0 e 1 do TLS ($\texttt{N} = 2$), por exemplo, e a aplicação cliente faz uma requisição (através das APIs do sistema operacional) a uma dessas posições, como à posição zero, a chamada de sistema é modificada para alvejar a posição deslocada em `N`, ou seja, a posição dois. Na prova de conceito desenvolvida, o valor de N é definido manualmente pelo analista de segurança. No entanto, é possível determinar N de maneira automática em tempo de execução, através da verificação das posições do TLS sendo utilizadas. A abordagem manual foi adotada na contramedida proposta por ter implementação mais simples e pelo valor de N não variar entre as execuções do *framework* DBI em uma mesma versão de sistema operacional.

Essa abordagem conta como principal desvantagem a redução efetiva do número total de posições do TLS disponíveis para a aplicação cliente utilizar, uma vez que determinado número de posições está sendo utilizado pelo *framework* DBI. Por este motivo, em condições de *arms race* - quando o *malware* tenta detectar a contramedida a técnica anti-instrumentação - seria possível detectar esta contramedida através da contagem das posições disponíveis do *Thread Local Storage*.

Para realizar o controle das chamadas às APIs relacionadas ao TLS, foram utilizadas as funcionalidades *ReplaceFunctionSignature* do **`PinVMShield`**, provida pela super classe *PinWrapperWinAPI* para interceptar as chamadas de sistema e a função *PIN_CallApplicationFunction*, para fazer uma nova chamada de sistema com parâmetros modificados, quando necessário. Dessa forma, toda vez que uma das APIs sendo monitorada (*TlsAlloc*, *TlsSetValue*, *TlsGetValue*, *TlsFree*) é invocada, ocorre a intercepção da chamada, a aplicação da política de controle de posições do TLS e a realização da chamada de sistema modificada.

Dessa forma, o isolamento entre as posições do TLS alocadas e utilizadas pelo *framework* DBI é garantida logicamente, ou seja, não há a separação real. Tanto a ferramenta DBI, quanto a aplicação cliente continuam compartilhando a estrutura. No entanto, através do monitoramento ativo das interações com o *Thread Local Storage* e da aplicação da política de controle de posições foi possível impedir interações da

aplicação cliente com o *framework* DBI, através do TLS.

5.2.2 *Memory Fingerprinting* / Detecção de Cache de Código

Esta subseção descreve a implementação da contramedida contra a técnica de evasão de *frameworks* DBI *Memory Fingerprinting* / Detecção de Cache de Código através de uma análise da estrutura Cache de Código e de que como a técnica anti-instrumentação interage com ela.

5.2.2.1 Cache de Código

Como mostrado na Figura 2.1, que ilustra a arquitetura do *framework* DBI Pin, o Cache de Código, ou *Code Cache*, é um componente integrante de motor DBI dos *frameworks* de instrumentação binária dinâmica. Como explicado na subseção 2.3, os Caches de Código são estruturas essenciais para o funcionamento das soluções DBI, pois é neles que reside o código da aplicação que de fato será executado e informações de controle específicas inerentes a instrumentação, como a cópia dos registradores do processador. Durante o processo de instrumentação binária dinâmica, o *framework* DBI utiliza o compilador *just-in-time* para criar uma versão modificada do código da aplicação cliente. Esse código modificado é alocado nos Caches de Código e quando necessário, a ferramenta DBI desvia o fluxo de execução para esses segmentos de memória. Devido a essas interações, os segmentos de memória utilizados como Caches de Código possuem permissões de leitura, escrita e execução.

Como sugerido por Sun et al. (2016) e Polino et al. (2017), um processo sob análise de uma ferramenta DBI pode realizar uma busca no seu espaço de memória virtual a fim de identificar Caches de Código. Em sistemas operacionais Windows, isso pode ser feito através de APIs do sistema, como *VirtualQuery* ou *NtQueryVirtualMemory*. De acordo com Sun et al. (2016), a busca por Caches de Código pode ser baseada em dois modelos distintos. No primeiro, chamada de padrões de código ou dados, uma sequência de instruções que deveria ser única na memória pode ter duas ou mais ocorrências em memória devido às Caches de Código. Na segunda, chamada de artefatos de Cache de Código, busca-se o hexadecimal *0xFEEDBEAF* que é empregado pela ferramenta DBI para identificar a região de Cache de Código.

5.2.2.2 Contramedida

Para realizar a mitigação da técnica anti-instrumentação *Memory Fingerprinting* / Detecção de Cache de Código é necessário estabelecer uma política eficaz de identificação das regiões de memória virtual e controle de acesso às regiões de memória que são Caches de Código, estabelecendo-as como *endereços de memória protegidos*. Para esse fim, o controle de acesso a memória pode ser feito através da monitoração e interceptação das chamadas às APIs *VirtualQuery* e *NtQueryVirtualMemory*. Dessa forma, quando a aplicação cliente invoca uma dessas APIs, o *framework* DBI pode tomar medidas imediatamente antes ou depois da chamada de sistema. Para realizar o controle das

regiões de memória que são Caches de Código podem ser utilizadas políticas de *blacklist* e *whitelist*, onde se estabelece quais endereços de memória podem ser acessados ou não pela aplicação cliente.

De forma prática, na geração da prova de conceito da contramedida proposta foi adotada a política de *blacklist* para estabelecer a identificação das regiões de memória virtual que a aplicação pode interagir. A lista de endereços contempla as faixas de endereço utilizadas como Caches de Código pela ferramenta DBI, sendo, dessa forma, preenchida em tempo de execução pelo `PinVMShield` através da API do Pin *CODECACHE_AddTraceInsertedFunction* que permite que uma função seja notificada cada vez que um Cache de Código é criado. Na implementação proposta, a função notificada pela criação de um Cache de Código é a função que faz a gerência da lista de regiões de memória protegidas.

A política de *blacklist* foi adotada, em detrimento a de *whitelist*, devido a segunda ter se mostrado ineficaz na implementação proposta por Polino et al. (2017), não sendo capaz de mitigar todos os casos de Detecção de Cache de Código, como a detecção do hexadecimal *0xFEEDBEAF*. Ademais, a política de *whitelist* se torna mais custosa computacionalmente e de implementação mais difícil, uma vez que sempre que a aplicação cliente aloca um novo endereço de memória é necessário incluir esse endereço na *whitelist*. Ao contrário da *whitelist*, na *blacklist* só ocorre uma adição na lista quando um novo Cache de Código é criado pelo *framework* DBI.

Para realizar o controle de acesso a memória foram utilizadas as funcionalidades providas pela classe *PinWrapperWinAPI*, especificamente a função *ReplaceFunctionSignature* do `PinVMShield` para interceptar as chamadas às APIs *VirtualQuery* e *NtQueryVirtualMemory*. Assim, toda vez que uma dessas APIs do sistema operacional é invocada, imediatamente antes de sua execução, são verificados todos os seus parâmetros. Caso seja detectado que a chamada de sistema referência a um endereço pertencente a um Cache de Código, a chamada ao sistema operacional retorna o valor zero, indicando que a chamada de sistema falhou. Em uma condição de *arms race*, um excesso ou um padrão de chamadas de sistema como retorno zero, indicando falha, poderia ser utilizado pelo atacante para apontar a presença desta contramedida.

Através da geração em tempo de execução da lista de Caches de Código e do controle das interações entre aplicação cliente e segmentos de memória, foi possível realizar o isolamento lógico entre os Caches de Código e a aplicação sob análise, apesar de esses elementos residirem, de fato, no mesmo espaço de endereços de memória virtual.

5.2.3 Negligenciamento do *No-eXecute Bit* (nx)

Esta seção expande a subseção 3.2.1.8, apresentando conceitos do *No-eXecute Bit* (nx) e como age a técnica de anti-instrumentação baseada no Negligenciamento do *No-eXecute Bit* (nx) feito pelos *frameworks* DBI. Em seguida, é apresentada a contramedida propostas.

5.2.3.1 *No-eXecute Bit* (nx)

Como apresentado na seção X, *No-eXecute Bit* (nx) é uma tecnologia de marcação de página de memórias, utilizada em processadores, que impede a execução de dados como código. Nos processadores Intel é nomeada como *Execute Disable Bit* (XD), enquanto nos processadores AMD é chamada de *No-eXecute Bit* (nx), ou *NX bit*.

A nível de sistema operacional, quando fazem uso desse bit de execução, recebem outras denominações. No Windows, a funcionalidade de marcação de páginas de memória como executável, ou não, é conhecida como *Data Execution Prevention* (DEP), no entanto é referenciada também como *No-Execute* (NX) (Microsoft, 2019a). Enquanto no Linux é conhecido apenas como *NX bit*.

Segundo Microsoft (2019b), a prevenção de execução de dados foca na mitigação de dados executados como instruções provenientes da *heap* padrão do processo e da pilha do processo. No entanto, não se limita a isso. Qualquer segmento de memória alocado que não possui permissões de execução, tem sua execução interrompida.

5.2.3.2 Contramedida

A contramedida para o Negligenciamento do *No-eXecute Bit* (nx) consiste em realizar uma implementação semelhante a dos sistemas operacionais de verificação do bit de execução antes da execução de qualquer código da aplicação, uma vez que esse comportamento está ausente nas ferramentas DBI. Nos sistemas operacionais Windows, esse mecanismos de verificação é conhecido como *Data Execution Prevention* (DEP) ou Prevenção de Execução de Dados Microsoft (2019b). Para o caso dos *frameworks* de instrumentação binária dinâmica, imediatamente antes do compilador *just-in-time* do *framework* ser invocado para construir as Caches de Código, deverá ser verificado se a região de memória, de onde é proveniente o código sendo utilizado como entrada do compilador, possui permissões de execução. Em caso negativo, da mesma forma que o sistema operacional faz, deverá ser lançada uma exceção com status *STATUS_ACCESS_VIOLATION*.

A implementação desta contramedida consiste em empregar o mesmo mecanismo utilizado para construir lista de Caches de Códigos apresentada na subseção 5.2.2 (nomeada como *blacklist*), a API do Pin *CODECACHE_AddTraceInsertedFunction*. Assim, toda vez que um Cache de Código é criado, ou reescrito pelo *framework* DBI, são verificadas as permissões (que podem ser uma combinação de leitura, escrita e execução) da região de memória do código que será alocado no Cache de Código em questão, através da API do *VirtualQuery* do sistema operacional. Quando é verificado que existe a permissão de execução, a ferramenta DBI continua sua execução normal. Caso contrário, a execução é suspensa temporariamente. O analista utilizando o *framework* DBI é notificado através de um *log* textual (arquivo de texto) podendo assim, tomar uma ação como continuar a execução da aplicação ou suspendê-la. Um comportamento alternativo ao proposto (e mais transparente) seria disparar uma exceção com o status *STATUS_ACCESS_VIOLATION*, imitando o comportamento normal do sistema operacional fora do *framework* DBI.

5.3 Eficácia e Desempenho da Contramedidas Propostas

Para determinar a eficácia das contramedidas propostas de maneira prática, foram necessárias provas de conceito (PoC) das técnicas anti-instrumentação. No entanto, os autores que as introduziram as propuseram apenas de maneira teórica, não havendo provas de conceito. Desta forma, através dos algoritmos presentes na literatura foram implementadas provas de conceito funcionais de cada uma das técnicas de evasão que tiveram contramedidas apresentadas na seção anterior. Esses artefatos encontran-se disponíveis online[2], além de estarem listados no apêndice A.

Em seu trabalho, Falcón and Riva (2012a) propuseram um *framework*, conhecido como `eXait`(Falcón and Riva, 2012b), para agrupar as técnicas anti-instrumentação apresentadas por eles. Nesta ferramenta, cada técnica anti-instrumentação é integrada ao *framework* como uma DLL, sendo carregada sob demanda, em tempo de execução, tornando-se uma ferramenta extensível e prática, uma vez que conta ainda com uma interface gráfica e um modo em console de comandos. Por isso, optou-se por integrar as provas de conceito desenvolvidas das técnicas anti-instrumentação a essa ferramenta. O *framework* encontra-se disponível online [3] sob uma licença BSD de uso.

Já as contramedidas propostas tiveram suas provas de conceito incorporadas na ferramenta `PinVMShield` (Rodriguez et al., 2016) para a validação de suas eficácias. Desta forma, o cenário de testes consiste no uso do `PinVMShield`, com as contramedidas habilitadas, sendo executado em conjunto com as provas de conceito das técnicas anti-instrumentação no `eXait`.

Como ambiente de experimentação, foi utilizado um ambiente virtualizado sobre um processador KVM de 8 cores 3.41 GHz, com 16GB de memória RAM. A máquina virtual utilizou o sistema operacional Windows 7 SP1 x64 com o Intel Pin 2.14-71313 (versão mais recente na versão 2.x) e compilador Microsoft 32bit C/C++ v16 para a compilação da ferramenta de *benchmark* SPEC2006. Uma vez que o sistema operacional Windows 7 foi o sistema alvejado pelas provas de conceito propostas na literatura e pela ferramenta `PinVMShield`, o mesmo foi adotado como sistema operacional do ambiente de testes.

Assim, como resultado, as técnicas Detecção por *Thread Local Storage* e *Memory Fingerpriting* / Detecção de Cache de Código foram mitigadas com sucesso. Similarmente, a técnica Negligenciamento do *No-eXecute Bit* (nx) pôde ser detectada em tempo de execução (e após a execução da aplicação), permitindo que o analista tome uma ação apropriada durante a execução da aplicação. Apesar do resultado positivo na mitigação das técnicas evasivas abordadas, após uma análise das contramedidas propostas, foram apontadas diversas condições de *arms race* (discutidas ao longo das subseções que apresentaram as contramedidas propostas) que posam como vulnerabilidades exploráveis por agentes maliciosos para revelar a presença dos *frameworks* DBI. Por exemplo, a contramedida proposta para a técnica anti-instrumentação Detecção

[2] `https://github.com/ailton07/eXait_Plugin_PinDetectionByDEPNeglect`, `https://github.com/ailton07/eXait_Plugin_CodeCacheDetectionByFEEDBEAF`, `https://github.com/ailton07/eXait_Plugin_PinDetectionByTLS`

[3] `https://github.com/ailton07/PinVMShield`

por *Thread Local Storage* (TLS) possui a desvantagem de manter o número reduzido de posições do TLS, uma vez que o *framework* DBI Pin reserva para seu uso determinado número de posições. Um hacker poderia equipar sua aplicação maliciosa com uma técnica evasiva que aloca todas as posições do TLS e as conta, a fim de apontar a presença do mecanismo de mitigação. Uma alternativa a implementação proposta, para contornar esse problema, seria realizar um isolamento lógico das posições de TLS utilizadas pelo Pin das posições TLS disponíveis para aplicação através de um algoritmo de cache em memória. Não obstante, as possibilidades de condições de *arms race* não foram esgotadas, podendo existir outras.

Para medir os impactos de desempenho do uso da ferramenta `PinVMShield` na aplicação cliente, foi utilizada a ferramenta de *benchmark* SPEC CPU2006 (CPU2006, 2006), uma suíte de *benchmarks* aceita como padrão por indústrias do setor de computação, que se destina a testar exaustivamente o processador, sistema de memória e compilador de um sistema e que é largamente utilizada para avaliar o desempenho de ferramentas DBI (Luk et al., 2005a; Arafa, 2017; Bruening et al., 2012).

Para o teste de desempenho, foi utilizado o conjunto de componentes de testes *SPECint2006* - assim como desenvolvedores do *framework* Pin Luk et al. (2005b) -, que possui um conjunto de doze (12) aplicações de testes diferentes utilizadas para se obter métricas de desempenho. Ressalta-se, no entanto, que apenas onze (11) são utilizáveis neste ambiente de testes, visto que a aplicação de teste *462.libquantum* do *SPEC* não é aplicável em ambientes com Microsoft Visual C++, como o deste trabalho.

O *SPEC* foi configurado para: (i) utilizar o tamanho de entrada de dados de referência; (ii) utilizar o modo não reportável, executando cada aplicação o número padrão de três vezes; e (iii) executar cada uma das onze aplicações de testes em conjunto com o *framework* Pin, a fim de estabelecer um piso de *overhead* inserido pela instrumentação binária dinâmica e em seguida, executar em conjunto com o `PinVMShield`. Enquanto executando o *SPEC* com o `PinVMShield`, funcionalidades não relacionadas às contramedidas apresentadas foram desabilitadas para evitar degradação de performance da aplicação.

A Tabela 5.1 sumariza os resultados obtidos para cada uma das aplicações de teste do *SPECint2006*. A primeira coluna lista os nomes das aplicações de *benchmark* do *SPEC*. A segunda coluna informa o tempo (em segundos) da execução do *framework* Pin com a ferramenta listada na primeira coluna. A terceira coluna apresenta o tempo (em segundos) da execução da aplicação listada na primeira coluna em conjunto com o `PinVMShield`. Finalmente, a quarta coluna exibe o *overhead* quando o `PinVMShield` é utilizado, em relação quando apenas o *framework* Pin é utilizado. Em média, o *overhead* inserido pelas contramedidas apresentadas foi de 59.73% com desvio padrão de 98.68%, o que indica que o *overhead* médio inserido varia significativamente em relação a aplicação sendo analisada.

Tabela 5.1: *Overhead* introduzido pelo `PinVMShield` com as contramedidas propostas.

Ferramenta de *Benchmark*	**Tempo de Instrumentação (s)**	**Tempo do *PinVMShield* (s)**	***Overhead* (%) *PinVMShield***
400.perlbench	484	534	10.3305
401.bzip2	560	562	0.3571
403.gcc	443	1150	159.5936
429.mcf	192	209	8.8541
445.gobmk	521	537	3.0710
456.hmmer	680	718	5.5882
458.sjeng	606	607	0.1650
464.h264ref	1069	4071	280.8231
471.omnetpp	294	819	178.5714
473.astar	339	326	-3.8348
483.xalancbmk	287	326	13.5888

5.4 Discussão

Este capítulo introduziu um conjunto de soluções capazes de mitigar as ameaças causadas por técnicas anti-instrumentação que exploram a superfície de ataque dos *frameworks* de instrumentação binária dinâmica e podem comprometer ambientes de análise que empregam tais ferramentas. Garantiu a aplicabilidade dessas soluções através de análises de eficácia e desempenho de provas de conceito e discutiu sobre as condições de *arms race* que podem levar as contramedidas apresentadas a falhar na proteção da transparência das ferramentas DBI.

Apesar de atestada a eficácia das contramedidas propostas em mitigar as técnicas anti-instrumentação que podem levar a fuga da instrumentação (ou a exploração da superfície de ataque introduzida pelas ferramentas DBI), diversas condições de *arms race*, consideradas como vulnerabilidades nos comportamentos das técnicas propostas, foram detectadas e podem ser apontadas a partir da engenharia reversa ou do estudo das contramedidas apresentadas.

As condições de *arms race* podem levar um *malware* evasivo a detectar o *framework* de instrumentação binária dinâmica, o que fere uma propriedade crítica, segundo Zhechev (2018), das ferramentas DBI para a análise de aplicações maliciosas, a transparência. No entanto, dadas as considerações das condições de *arms race*, as estruturas críticas do *framework* DBI foram corretamente protegidas da aplicação cliente, garantindo a propriedade de isolamento das ferramentas DBI. Dessa forma, foi possível reduzir a superfície de ataque explorável dos *frameworks* de instrumentação binária dinâmica através da mitigação das técnicas evasivas que exploram as vulnerabilidades desses *frameworks*, evitando ataques, antes viabilizados pelas ferramentas DBI, como a execução de código arbitrário (*Arbitrary Code Execution*). Assim, este trabalho resulta em uma resolução que difere da apresentada por Zhechev (2018), concluindo que os *frameworks* de instrumentação binária dinâmica não tornam necessariamente os ambientes que os

utilizam menos seguros e podem ser utilizados para aplicações de segurança, desde que observado o contexto de aplicação e consideradas as limitações de transparência e isolamento. Ademais este trabalho demonstra que é possível desenvolver soluções que garantem o isolamento e a integridade das ferramentas DBI.

Sob certa perspectiva, pode-se afirmar que enquanto as soluções propostas como contramedidas mitigam as técnicas anti-instrumentação presentes na literatura, eles permitem o surgimento de novas. Esse cenário evidencia a condição de *cabo de guerra* entre os profissionais de segurança e os desenvolvedores de aplicações maliciosas, além de elucidar o significado da expressão *arms race*, corrida armamentista (em tradução livre), onde ambos os lados desenvolvem ferramentas em ritmo acelerado para sobrepujar as do lado contrário.

Capítulo 6

Considerações Finais

Aplicações maliciosas capazes de detectar ambientes de análise são ameaças potenciais a ambientes virtuais, uma vez que podem se comportar como uma aplicação benigna a fim de evadir sistemas de detecção e classificação de *malwares*. De forma geral, elas podem ser adotadas em ameaças virtuais especialistas com alta capacidade de danos, como as *Advanced Persistent Threat* (APT) focadas em indivíduos ou ambientes específicos (por exemplo, industrias, empresas e setores governamentais) (Che, 2016).

No entanto, as técnicas anti-instrumentação estudadas nesta dissertação posam como riscos potenciais, mesmo entre os *malwares* evasivos, uma vez que podem ser utilizadas como vetores de ataque (meios disponíveis pelos quais um *hacker* pode realizar um ataque (Drake et al., 2014). Desenvolvedores de aplicações maliciosas podem infectar ambientes de análise que utilizam ferramentas DBI através da exploração de vulnerabilidades dessas ferramentas, o que é uma evolução do objetivo inicial das técnicas de evasão: impedir a detecção de uma aplicação maliciosa como tal (Rodriguez et al., 2016; Polino et al., 2017; Balzarotti et al., 2010).

A fim de determinar se era possível reduzir a superfície de ataque explorável introduzida pelos *frameworks* DBI, demonstrada por Sun et al. (2016) e Zhechev (2018), esta dissertação inicialmente analisou as técnicas anti-instrumentação que abusam das vulnerabilidades dos *frameworks* de instrumentação binária dinâmica e não possuem contramedidas na literatura. Para isso, foram revisadas todas as técnicas anti-instrumentação presentes na bibliografia relacionada e analisadas sob a ótica da fuga da instrumentação binária dinâmica, definida por Sun et al. (2016). Uma vez que as técnicas evasivas selecionadas foram propostas apenas teoricamente, foram desenvolvidos artefatos de código, em forma de provas de conceito (PoC) dessas técnicas, para fins de validação das contramedidas propostas.

Em seguida, foram desenvolvidas soluções de mitigação e implementados em forma de provas de conceito contra as técnicas anti-instrumentação selecionadas. As contramedidas propostas tiveram PoCs desenvolvidas sobre a ferramenta `PinVMShield`, uma solução desenvolvida para mitigar técnicas de evasão diversas. Ambas, provas de conceito das técnicas evasivas e das contramedidas foram avaliadas em um ambiente virtualizado para atestar suas eficácias. Após os testes de eficácia, foi realizada uma análise do

overhead temporal introduzido pelas contramedidas desenvolvidas (aproximadamente 60% em média).

6.1 Conclusões

Esta dissertação focou em descobrir se as contramedidas apresentadas afetavam a superfície de ataque explorável introduzida pelas ferramentas DBI. Diferente do trabalho apresentado por Zhechev (2018), esta dissertação demonstrou que os *frameworks* DBI são capazes de manter a propriedade de isolamento, desde que as vulnerabilidades apresentadas pelos mesmos sejam tratadas. Quanto a transparência, nas soluções apresentadas por essa dissertação, de fato existiram casos onde não foi possível garanti-la, devido a condições de *arms race* que permitem que desenvolvedores de aplicações maliciosas detectem as ferramentas DBI através de comportamentos e características das contramedidas propostas. No entanto, Bruening et al. (2012) em seu trabalho sobre a transparência da instrumentação binária dinâmica apontou a existência de um *tradeoff* entre desempenho (*overhead* inserido) e transparência. Assim, devido a isso, existiriam casos onde a total transparência poderia não ser alcançável na prática.

De forma prática, esta dissertação demonstrou que os *frameworks* de instrumentação binária dinâmica podem ser utilizados para fins de segurança de sistemas, desde que observados os requisitos de segurança no contexto sendo empregado (como o isolamento, a transparência e o desempenho) e que devidamente equipados com as ferramentas apropriadas (como contramedidas anti-instrumentação e anti-evasão). Ressalta-se também que as propriedades de transparência e isolamento de outras ferramentas de detecção e classificação de *malwares* são constantemente alvejadas por desenvolvedores de aplicações maliciosas, como apresentado por Egele et al. (2012) e Bulazel and Yener (2017), explicitando que trata-se de um *cabo de guerra* entre *hackers* e profissionais de segurança. Não obstante, é necessário observar o aumento em número e sofisticação das técnicas evasivas e responder adequadamente através de aperfeiçoamentos nas ferramentas de análise e mitigação das ameaças emergentes.

6.2 Dificuldades Encontradas

Para atingir o objetivo proposto nesta dissertação, foi estipulado que era preciso identificar as técnicas de evasão anti-instrumentação que exploram a superfície de ataque inserida pelos *frameworks* DBI, dentre o universo de técnicas apresentado na literatura. Além da pouca quantidade publicada, comumente os autores não fornecem código fonte, algoritmos ou provas de conceito das implementações das técnicas de evasão e das contramedidas sendo propostas por eles. Esses artefatos poderiam auxiliar na reprodução dos cenários apresentados por esses autores e na proposição de novas contramedidas. Desta forma, um desafio foi desenvolver algoritmos e implementações de técnicas anti-instrumentação a partir somente da descrição da estrutura alvejada pela técnica. Conforme apresentado na seção 5.3 e seguindo o espírito do livre acesso a ciência, o código fonte dessas provas de conceito encontra-se disponível online publicamente,

assim como das contramedidas desenvolvidas.

Outra limitação percebida na literatura é que, embora haja um grande esforço de profissionais de segurança e pesquisadores em desenvolver contramedidas para as técnicas anti-instrumentação, os trabalhos científicos apresentam discussões escassas das soluções ou até mesmo inexistente acerca dos impactos de desempenho dessas contramedidas sobre os *frameworks* DBI e dos ambientes de teste de eficácia e desempenho. Ademais, nenhum trabalho até este, discutiu sobre as vulnerabilidades das contramedidas propostas, condições de *arms race* ou fez uma análise dos casos em que as contramedidas não são eficazes.

Além disso, apesar da quantidade de técnicas anti-instrumentação, 33, até este trabalho não havia uma revisão suficientemente ampla para exaurir todas as técnicas apresentadas na literatura. O mesmo pode ser dito sobre as classificações sugeridas pelos autores (Rodriguez et al., 2016; Sun et al., 2016; Polino et al., 2017; Zhechev, 2018). Nenhuma delas foi capaz de classificar sistematicamente de maneira independente da ferramenta DBI alvejada e considerando as naturezas direta e indireta das técnicas anti-instrumentação.

Por fim, notou-se durante a revisão da literatura uma predominância dos trabalhos apresentados em formato de "palestras" em conferências de segurança relacionadas à indústria. Assim, seja pela natureza do evento ou do próprio formato de trabalho, tais trabalhos carecem de formalismo, uma vez que possuem pouco conteúdo textual quando comparados à artigos científicos.

6.3 Tópicos em Aberto

Apesar dos esforços dos desenvolvedores de ferramentas de segurança, existem técnicas anti-instrumentação que não possuem contramedidas e ameaçam a transparência das ferramentas DBI, posando como um desafio aos profissionais de segurança. Pode-se destacar, por exemplo, como problemas em aberto:

- A detecção de ferramentas de análise dinâmica e ambientes virtualizados através do *overhead* temporal inserido; Problema que afeta essencialmente todas as ferramentas de análise dinâmica;

- Como discutido nos capítulos anteriores, o *cabo de guerra* entre desenvolvedores de aplicações maliciosas e profissionais de segurança irá gerar novas e mais sofisticadas técnicas anti-instrumentação, o que irá demandar a proposição de novas contramedidas;

- Uma vez que ainda não foi possível, segundo a literatura, garantir a total transparência das ferramentas DBI em relação às aplicações sendo instrumentadas - *tradeoff* entre desempenho e transparência, discutido na seção 6.1 -, análises e pesquisa sobre se os *frameworks* DBI são a opção ótima para a análise de software não seguro e/ou potencialmente malicioso ainda é um ponto em aberto;

- Apesar das contramedidas propostas terem suas eficácia atestadas através de testes com provas de conceito das técnicas anti-instrumentação, elas ainda carecem de testes em um cenário mais próximo do mundo real, com aplicações maliciosas autênticas que empregam tais comportamentos evasivos;

- Por fim, a literatura não apresenta ainda um consenso, em termos de comparação, quanto a eficácia das soluções de proteção de *frameworks* DBI (Rodriguez et al., 2016; Polino et al., 2017). Da mesma forma que não existe uma comparação quanto aos impactos de desempenho (*overhead* temporal inserido) de tais soluções.

Referências Bibliográficas

(2016). Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. In *IFIP Advances in Information and Communication Technology*, volume 471, pages 323–336.

Afianian, A., Niksefat, S., Sadeghiyan, B., and Baptiste, D. (2018). Malware Dynamic Analysis Evasion Techniques: A Survey.

Anley, C., Heasman, J., Lindner, F., and Richarte, G. (2011). *The shellcoder's handbook: discovering and exploiting security holes*. John Wiley & Sons.

Arafa, P. (2017). *Time-Aware Dynamic Binary Instrumentation*. PhD thesis, University of Waterloo.

AV-TEST GmbH (2018). The AV-TEST Security Report 2017/2018. [Online; https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2017-2018.pdf]. Acessado em 18.12.2018.

Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., and Vigna, G. (2010). Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Bruening, D., Duesterwald, E., and Amarasinghe, S. (2001). Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.

Bruening, D., Zhao, Q., and Amarasinghe, S. (2012). Transparent dynamic instrumentation. *ACM SIGPLAN Notices*, 47(7):133–144.

Bulazel, A. and Yener, B. (2017). A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, page 2. ACM.

Carpenter, M., Liston, T., and Skoudis, E. (2007). Hiding Virtualization from Attackers and Malware. *IEEE Security & Privacy*, 5(3):62–65.

Cerrudo, C. (2005). Hacking Windows Internals. *Black Hat EU*.

Computer Security Resource Center (2018). CSRC - Glossary - Application Whitelist.

Corporation, I. (2011). Intel ® 64 and IA-32 Architectures Software Developer ' s Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. *System*, 3(253665).

CPU2006, S. (2006). Standard Performance Evaluation Corporation. [Online; https://www.spec.org/cpu2006/].

Deng, Z., Zhang, X., and Xu, D. (2013). SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization. *Annual Computer Security Applications Conference*, pages 289–298.

Die.net (2018). setrlimit(2): get/set resource limits - Linux man page. [Online; https://linux.die.net/man/2/setrlimit].

Drake, J. J., Lanier, Z., Mulliner, C., Fora, P. O., and Ridley, S. A. (2014). *Android Hacker's Handbook*.

Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv.*, 44(2):6:1—-6:42.

Eilam, E. and Chikofsky, E. J. (2005). *Reversing: Secrets of Reverse Engineering*.

Ekenstein, G. and Norrestam, D. (2017). Classifying evasive malware. Master's thesis, Lund University.

Falcón, F. and Riva, N. (2012a). Dynamic Binary Instrumentation Frameworks: I know you're there spying on me.

Falcón, F. and Riva, N. (2012b). eXait | Core Security. [Online; https://www.secureauth.com/labs/open-source-tools/exait].

Gilboy, M. R. (2016). Fighting Evasive Malware With Dvasion. Master's thesis, University of Maryland.

Greamo, C. and Ghosh, A. (2011). Sandboxing and Virtualization: Modern Tools for Combating Malware. *IEEE Security & Privacy*, 9(2):79–82.

Hron, M. and Jermář, J. (2014). SafeMachine malware needs love, too. *Virus Bulletin, VB2014 Seattle*.

Intel Corporation (2013). Pin: Intel's Dynamic Binary Instrumentation Engine. In *International Symposium on Code Generation and Optimization*.

Intel Corporation (2017a). Pin: IMG: Image Object.

Intel Corporation (2017b). Pin: RTN: Routine Object.

Intel Corporation (2017c). Pin: TRACE: Single entrance, multiple exit sequence of instructions.

Kaspersky Lab (2017). Kaspersky Lab detects 360,000 new malicious files daily – up 11.5% from 2016. [Online; https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily]. Acessado em 18.12.2018.

Kirat, D., Vigna, G., and Kruegel, C. (2011). Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM.

Kirat, D., Vigna, G., and Kruegel, C. (2014). Barecloud: Bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, San Diego, CA. USENIX Association.

Kolbitsch, C., Kirda, E., and Kruegel, C. (2011). The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296. ACM.

Kulakov, Y. (2017). MazeWalker - Enriching Static Malware Analysis. [Online; https://recon.cx/2017/montreal/resources/slides/RECON-MTL-2017-MazeWalker.pdf]. Acessado em 18.12.2018.

Kumar, A. V., Vishnani, K., and Kumar, K. V. (2012). Split Personality Malware Detection and Defeating in Popular Virtual Machines. In *Proceedings of the 5th International Conference on Security of Information and Networks (SIN)*, pages 20–26. ACM.

Ligh, M. H., Case, A., Levy, J., and Walters, A. (2014). *The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory*. Wiley, Indianapolis, IN. OCLC: ocn885319205.

Linux Programmer's Manual (2018). proc(5) - Linux manual page. [Online; http://man7.org/linux/man-pages/man5/proc.5.html].

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005a). Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA. ACM.

Luk, C.-K., Ed, B. C., Hi, F. C. G., Rs, E. D. Q., Tu, A., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005b). Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05*, 40(6):190.

Manadhata, P. K. (2008). *An Attack Surface Metric*. PhD thesis, Carnegie Mellon University.

Mariani, S., Fontana, L., and Gritti, F. (2016). PinDemonium: a DBI-based generic unpacker for Windows executables. [Online; https://www.blackhat.com/html/webcast/12152016-pindemonium-a-dbi-based-generic-unpacker-for-windows-executables.html]. Acessado em 18.12.2018.

McAfee Labs (2017). McAfee Labs Threats Report: June 2017. [Online; https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2017.pdf].

Microsoft (2017a). kuser | Microsoft Docs. [Online; https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-kuser].

Microsoft (2017b). Virtual Address Space (Windows). [Online; https://msdn.microsoft.com/pt-br/library/windows/desktop/aa366912(v=vs.85).aspx].

Microsoft (2018a). Critical System Services. [Online; https://msdn.microsoft.com/en-us/library/windows/desktop/aa373646(v=vs.85).aspx].

Microsoft (2018b). Definições de argumento. [Online; https://msdn.microsoft.com/pt-br/library/88w63h9k.aspx].

Microsoft (2018c). Event Objects (Windows). [Online; https://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx].

Microsoft (2018d). File Handles (Windows). [Online; https://msdn.microsoft.com/pt-br/library/windows/desktop/aa364225(v=vs.85).aspx].

Microsoft (2018e). NtOpenProcess function. [Online; https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-ntopenprocess].

Microsoft (2018f). NtQuerySystemInformation function (Windows). [Online; https://msdn.microsoft.com/en-us/library/windows/desktop/ms724509(v=vs.85).aspx].

Microsoft (2018g). Process Handles and Identifiers (Windows). [Online; https://msdn.microsoft.com/pt-br/library/windows/desktop/ms684868(v=vs.85).aspx].

Microsoft (2018h). Running 32-bit Applications (Windows). [Online; https://msdn.microsoft.com/en-us/library/windows/desktop/aa384249(v=vs.85).aspx].

Microsoft (2018i). Thread Local Storage. [Online; https://msdn.microsoft.com/en-us/library/windows/desktop/ms686749(v=vs.85).aspx].

Microsoft (2018j). VirtualQuery function. [Online; https://msdn.microsoft.com/pt-br/library/windows/desktop/aa366902(v=vs.85).aspx].

Microsoft (2019a). DEP/NX Protection - Windows applications | Microsoft Docs. [Online; https://docs.microsoft.com/en-us/windows/desktop/win7appqual/dep-nx-protection].

Microsoft (2019b). Uma descrição detalhada do recurso DEP (Prevenção de execução de dados) no Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005 e Windows Server 2003. [Online; https://support.microsoft.com/pt-br/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in].

Nethercote, N. (2004). *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy*. PhD thesis, University of Cambridge.

Nethercote, N. and Seward, J. (2003). Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, pages 47–69.

Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontata, L., Gritti, F., and Zanero, S. (2017). Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*.

Qiu, J., Yadegari, B., Johannesmeyer, B., Debray, S., and Su, X. (2014). A Framework for Understanding Dynamic Anti-Analysis Defenses. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*.

Rodríguez, R. J., Artal, J. A., Merseguer, J., and Artal, J. A. (2014). Performance Evaluation of Dynamic Binary Instrumentation Frameworks | R.J. Rodriguez, J.A. Artal, J. Merseguer.

Rodriguez, R. J., Gaston, I. R., and Alonso, J. (2016). Towards the detection of isolation-aware malware. *IEEE Latin America Transactions*, 14(2):1024–1036.

Stamatogiannakis, M., Groth, P., and Bos, H. (2015). Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In Ludäscher, B. and Plale, B., editors, *Provenance and Annotation of Data and Processes*, pages 155–167, Cham. Springer International Publishing.

Sun, K., Li, X., and Ou, Y. (2016). Break Out of The Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation. Black Hat Asia.

Tanenbaum, A. S. (2008). *Sistemas operacionais modernos*. Pearson Prentice Hall, São Paulo. OCLC: 457537581.

Vishnani, K., Pais, A. R., and Mohandas, R. (2011). Detecting & Defeating Split Personality Malware. In *Proocedings of the 5th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, pages 7–13.

Zhechev, Z. (2018). Security Evaluation of Dynamic Binary Instrumentation Engines. Master's thesis, Departmente of Informatics Technical University of Munich.

Apêndice A

Provas de Conceito das Técnicas Anti-Instrumentação

Os projetos completos das provas de coneito das técnicas de evasão apresentadas neste apêndice encontram-se disponíveis online[1].

A.1 Negligenciamento do No-eXecute Bit (nx)

```
// PinDetectionByDEPNeglect.cpp : Defines the entry point for the
    console application.
//
// Basta procurar por call [0x0130573C] no debug
#include <stdio.h>
#include <Windows.h>

#define ASSEMBLY_SIZE 4
const unsigned char assembly [ASSEMBLY_SIZE] = {0x90 , 0x90 , 0
   x90 , 0xC3 } ; // NOP, NOP, NOP, RET

static bool pinDetected = 0;

// ---------- start eXait header
#define DllExport extern "C" __declspec(dllexport)
DllExport char* GetPluginName(void);
DllExport char* GetPluginDescription(void);
DllExport int DoMyJob(void);

char* GetPluginName(void)
{
        static char PluginName[] = "PinDetectionByDEPNeglect";
```

[1] https://github.com/ailton07/eXait_Plugin_PinDetectionByDEPNeglect,
https://github.com/ailton07/eXait_Plugin_CodeCacheDetectionByFEEDBEAF,
https://github.com/ailton07/eXait_Plugin_PinDetectionByTLS

```
21         return PluginName;
22 }
23
24 char* GetPluginDescription(void)
25 {
26         static char MyDescription[] = "This plugin exploit a
              failure in page permissions treatment";
27         return MyDescription;
28 }
29 // ---------- end of eXait header modifications
30
31
32 int main(int argc, char** argv) {
33         printf ("Address of assembly variable: 0x%x\n", assembly)
              ;
34         // Tentativa 2
35         LPVOID virtualAddress;
36         //virtualAddress = VirtualAlloc(NULL, ASSEMBLY_SIZE,
              MEM_COMMIT,PAGE_EXECUTE_READWRITE);
37         // PAGE_READWRITE: Enables read-only or read/write access
               to the committed region of pages. If Data Execution
              Prevention is enabled, attempting to execute code in
              the committed region results in an access violation.
38         // https://docs.microsoft.com/pt-br/windows/desktop/
              Memory/memory-protection-constants
39         virtualAddress = VirtualAlloc(NULL, ASSEMBLY_SIZE,
              MEM_COMMIT,PAGE_READWRITE);
40         if (virtualAddress != NULL)
41         {
42                 CopyMemory(virtualAddress, assembly,
                      ASSEMBLY_SIZE);
43
44                 __try {
45                         __asm{
46                                 call virtualAddress
47                         }
48                         pinDetected = 1;
49                 }
50                 __except(EXCEPTION_EXECUTE_HANDLER){
51                         pinDetected = 0;
52                 }
53         }
54         printf("pinDetected = %d", pinDetected);
55         getchar();
56         return 0;
57 }
58
59 int DoMyJob(void)
60 {
```

```
61         printf ("Address of assembly variable: 0x%x\n", assembly)
              ;
62         // Tentativa 2
63         LPVOID virtualAddress;
64         //virtualAddress = VirtualAlloc(NULL, ASSEMBLY_SIZE,
              MEM_COMMIT,PAGE_EXECUTE_READWRITE);
65         virtualAddress = VirtualAlloc(NULL, ASSEMBLY_SIZE,
              MEM_COMMIT,PAGE_READWRITE);
66         if (virtualAddress != NULL)
67         {
68                 CopyMemory(virtualAddress, assembly,
                      ASSEMBLY_SIZE);
69
70                 __try {
71                         __asm{
72                                 call virtualAddress
73                         }
74                         pinDetected = 1;
75                 }
76                 __except(EXCEPTION_EXECUTE_HANDLER){
77                         pinDetected = 0;
78                 }
79         }
80         return pinDetected;
81 }
```

A.2 Memory Fingerpriting / Detecção de Cache de Código

```
1  // MemUpdateMapInformations.cpp : Defines the entry point for the
       console application.
2  // https://github.com/x64dbg/x64dbg/blob/598
      fc65ea0a0ccf9dad1a880355718529730614a/src/dbg/memory.cpp
3  //
4
5  //#include <MemUpdateMapInformations.h>
6  #include "MemUpdateMapInformations.h"
7
8  std::vector<MEMPAGE> GetPageVector()
9  {
10          std::vector<MEMPAGE> pageVector;
11          // Aloca um espaco previamente
12          pageVector.reserve(200); //TODO: provide a better
               estimate
13     {
14                 SIZE_T numBytes = 0;
15         DWORD pageStart = 0;
16         DWORD allocationBase = 0;
```

```
17
18              do
19      {
20                      // Query memory attributes
21          MEMORY_BASIC_INFORMATION mbi;
22                      memset(&mbi, 0, sizeof(mbi));
23
24                      // numBytes = VirtualQueryEx(
                            fdProcessInfo->hProcess, (LPVOID)
                            pageStart, &mbi, sizeof(mbi));
25                      numBytes = VirtualQuery((LPVOID)pageStart
                            , &mbi, sizeof(mbi));
26                      // Only allow pages that are committed/
                            reserved (exclude free memory)
27          if(mbi.State != MEM_FREE)
28          {
29                              auto bReserved = mbi.State ==
                                    MEM_RESERVE; //check if the
                                    current page is reserved.
30                              auto bPrivateType = mbi.Type ==
                                    MEM_PRIVATE;
31
32                              auto bPrevReserved = pageVector.
                                    size() ? pageVector.back().mbi
                                    .State == MEM_RESERVE : false;
                                     //back if the previous page
                                    was reserved (meaning this one
                                     won't be so it has to be
                                    added to the map)
33                              // Only list allocation bases,
                                    unless if forced to list all
34                              // if(bListAllPages || bReserved
                                    || bPrevReserved ||
                                    allocationBase != DWORD(mbi.
                                    AllocationBase))
35                              if(bReserved || bPrevReserved ||
                                    allocationBase != DWORD(mbi.
                                    AllocationBase))
36              {
37                                      // Set the new allocation
                                            base page
38                  allocationBase = DWORD(mbi.AllocationBase);
39
40                                      MEMPAGE curPage;
41                  memset(&curPage, 0, sizeof(MEMPAGE));
42                  memcpy(&curPage.mbi, &mbi, sizeof(mbi));
43
44                                      if(bPrivateType && (mbi.
                                            AllocationProtect ==
```

```
                                                RWE))
45                      {
46                          /* if(DWORD(curPage.mbi.BaseAddress) !=
                               allocationBase)
47                               sprintf_s(curPage.info, "DBG Reserved
                                     RWE (%p)", allocationBase);
48                           else
49                               sprintf_s(curPage.info, "DBG Reserved
                                     RWE");*/
50                                                   sprintf_s(curPage
                                                         .info, "DBG
                                                         Reserved (%p)
                                                         (0x%x)",
                                                         allocationBase
                                                         , mbi.
                                                         AllocationProtect
                                                         );
51                      }
52                                          pageVector.push_back(
                                                curPage);
53                                  }
54                          }
55
56                          // Calculate the next page start
57              DWORD newAddress = DWORD(mbi.BaseAddress) + mbi.
                    RegionSize;
58
59              if(newAddress <= pageStart)
60                  break;
61
62                          pageStart = newAddress;
63
64                  } while(numBytes);
65          }
66
67          return pageVector;
68
69  }
```

A.3 Detecção por Thread Local Storage (TLS)

```
1
2  #include <windows.h>
3  #include <stdio.h>
4
5  #define _LIMIAR 3
6
7  // ---------- start of eXait header modifications
8  #define DllExport extern "C" __declspec(dllexport)
```

```
9  DllExport char* GetPluginName(void);
10 DllExport char* GetPluginDescription(void);
11 DllExport int DoMyJob(void);
12
13 char* GetPluginName(void)
14 {
15         static char PluginName[] = "Detect Pin by TLS";
16         return PluginName;
17 }
18
19 char* GetPluginDescription(void)
20 {
21         static char MyDescription[] = "This plugin implements a
              search function to search for TLS entries.";
22         return MyDescription;
23 }
24 // ---------- end of eXait header modifications
25
26
27 VOID ErrorExit (LPSTR lpszMessage)
28 {
29         fprintf(stderr, "%s\n", lpszMessage);
30         ExitProcess(0);
31 }
32
33 int count32tls()
34 {
35         printf("\t== count32tls ==\n");
36         int quantidade = 0;
37         DWORD dwTlsIndex;
38         LPVOID lpvData;
39         int value;
40
41         for(dwTlsIndex = 0; dwTlsIndex < 32; dwTlsIndex++)
42         {
43
44                 lpvData = TlsGetValue(dwTlsIndex);
45                 value = reinterpret_cast<int>(lpvData);
46
47                 // printf("TLS[%d] = lpvData=%lx\n", dwTlsIndex,
                      value);
48                 if(value != 0) {
49                         quantidade++;
50                         // printf("TLS[%d] = lpvData igual a zero
                              \n", dwTlsIndex);
51                 }
52
53         }
54         return quantidade;
```

```
55 }
56
57 int DoMyJob(void)
58 {
59         int quantidade = 0;
60
61         quantidade = count32tls();
62
63         printf("Quantidade de slots do TLS ativos: %d\n",
               quantidade);
64
65         if(quantidade > _LIMIAR)
66         {
67                 printf("PIN DETECTED\n");
68                 return 1;
69         }
70         else
71         {
72                 printf("Pin not detected\n");
73                 return 0;
74         }
75
76         //system("pause");
77         return 0;
78 }
```

Apêndice B

Provas de Conceito das Contramedidas

Os projetos completos das provas de coneito das contramedidas apresentadas neste apêndice encontram-se disponíveis online[1].

B.1 Negligenciamento do No-eXecute Bit (nx)

Trecho do arquivo PinVMShield/PinVMShield.cpp referente à contramedida Negligenciamento do No-eXecute Bit (nx)

```

int main(int argc, char *argv[])
{
        // Initialize pin & symbol manager
    PIN_InitSymbols();
    PIN_Init(argc,argv);

        // Register Image to be called to instrument functions.
    IMG_AddInstrumentFunction(Image, 0);

        CODECACHE_AddTraceInsertedFunction(WatchTraces, 0);

        //INS_AddInstrumentFunction(Instruction , 0);

        printMessage("\t-> PIN_StartProgram();\n");

    PIN_StartProgram(); // Never returns

    return 0;
}

```

[1] https://github.com/ailton07/PinVMShield

```
22 // https://github.com/jingpu/pintools/blob/master/source/tools/
      CacheClient/insertDelete.cpp
23 /**
24  Preenche uma lista com as informacoes sobre os CodeCaches
25  This method obtain information about codecache address.
26  */
27 VOID WatchTraces(TRACE trace, VOID *v)
28 {
29     char textToPrint[4096];
30         ADDRINT orig_pc = TRACE_Address(trace);
31         ADDRINT codeCacheAddress = TRACE_CodeCacheAddress(trace);
32 
33         // Check permissions of memory page
34         // Detect PinDetectionByDEPNeglect technique
35         WINDOWS::MEMORY_BASIC_INFORMATION mbi;
36         memset(&mbi, 0, sizeof(mbi));
37         WINDOWS::SIZE_T numBytes = WINDOWS::VirtualQuery((WINDOWS
              ::LPVOID)orig_pc, &mbi, sizeof(mbi));
38         if (numBytes!= 0 && mbi.Protect != 0x20)
39         {
40                 sprintf(textToPrint, "\n[TRACE_Address] 0x%x ",
                      orig_pc);
41                 printMessage(TEXT(textToPrint));
42 
43                 sprintf(textToPrint, "\t[TRACE_CodeCacheAddress]
                      0x%x ", codeCacheAddress);
44                 printMessage(TEXT(textToPrint));
45 
46                 sprintf(textToPrint, "\t[TRACE_Size] 0x%x ",
                      TRACE_Size(trace));
47                 printMessage(TEXT(textToPrint));
48 
49                 sprintf(textToPrint, "\t[TRACE_CodeCacheSize] 0x%
                      x", TRACE_CodeCacheSize(trace));
50                 printMessage(TEXT(textToPrint));
51 
52                 sprintf(textToPrint, "\t[TRACE_Address] 0x%x [
                      Permissions] 0x%x\n", orig_pc, mbi.Protect);
53                 printMessage(TEXT(textToPrint));
54         }
55 
56 
57         for(int i = 0 ; i < addrintVector.size(); i++)
58         {
59                 if ((addrintVector[i] == codeCacheAddress) ||
60                         (codeCacheAddress > addrintVector[i]) &&
61                         (codeCacheAddress < addrintVector[i] +
                              codeCacheBlockSize))
62                 {
```

```
63                              return;
64                    }
65          }
66
67          sprintf(textToPrint, "\n[TRACE_Address] 0x%x ", orig_pc);
68          printMessage(TEXT(textToPrint));
69
70          sprintf(textToPrint, "\t[TRACE_CodeCacheAddress] 0x%x ",
                codeCacheAddress);
71          printMessage(TEXT(textToPrint));
72
73          sprintf(textToPrint, "\t[TRACE_Size] 0x%x ", TRACE_Size(
                trace));
74          printMessage(TEXT(textToPrint));
75
76          sprintf(textToPrint, "\t[TRACE_CodeCacheSize] 0x%x \n",
                TRACE_CodeCacheSize(trace));
77          printMessage(TEXT(textToPrint));
78
79          addrintVector.push_back(codeCacheAddress);
80 }
```

B.2 Memory Fingerpriting / Detecção de Cache de Código

```
1  /*
2  * @filename             WrapperPageQueryFeedbeaf.h
3  * @version                      1.1, Oct 2018
4  * @author                       A. da Silva
5  * @description
6  *
7  * This program is free software: you can redistribute it and/or
       modify
8  * it under the terms of the GNU General Public License as
       published by
9  * the Free Software Foundation, either version 3 of the License,
       or
10 * any later version.
11 *
12 * This program is distributed in the hope that it will be useful,
13 * but WITHOUT ANY WARRANTY; without even the implied warranty of
14 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15 * GNU General Public License for more details.
16 *
17 * You should have received a copy of the GNU General Public
       License
```

```
18 * along with this program.  If not, see <http://www.gnu.org/
      licenses/>.
19 */
20
21 #include "PinWrapperWinAPI.h"
22
23
24 class WrapperPageQueryFeedbeaf: public PinWrapperWinAPI {
25 private:
26         /**
27         Wrapper of VirtualQuery Windows APIs. Spoofs the return
              if needed
28         before returning to the caller
29         @param lpAddress [in, optional] a pointer to the base
              address of the region of pages to be queried.
30         @param lpBuffer [out] a pointer to a
              MEMORY_BASIC_INFORMATION structure in which
              information about the specified page range is returned
              .
31         @param dwLength [in] the size of the buffer pointed to by
               the lpBuffer parameter, in bytes.
32         @return the return value is the actual number of bytes
              returned in the information buffer.
33         For more information, check https://msdn.microsoft.com/en
              -us/library/windows/desktop/aa366902(v=vs.85).aspx
34         */
35         static WINDOWS::SIZE_T myPageQuery(AFUNPTR orig,
36                 WINDOWS::LPCVOID lpAddress,
37                 WINDOWS::PMEMORY_BASIC_INFORMATION lpBuffer,
38                 WINDOWS::SIZE_T dwLength,
39                 CONTEXT *ctx,
40                 bool isUnicode) {
41                         WINDOWS::SIZE_T retVal = 0;
42                         bool isCodeCache = false;
43                         unsigned long* baseAddress = (unsigned
                              long *) lpAddress;
44
45                         PIN_CallApplicationFunction(ctx,
                              PIN_ThreadId(),
46                                 CALLINGSTD_STDCALL, orig, NULL,
47                                 PIN_PARG(WINDOWS::SIZE_T), &
                                      retVal,
48                                 PIN_PARG(WINDOWS::LPCVOID),
                                      baseAddress,
49                                 PIN_PARG(WINDOWS::
                                      PMEMORY_BASIC_INFORMATION),
                                      lpBuffer,
50                                 PIN_PARG(WINDOWS::SIZE_T),
                                      dwLength,
```

```
51                                  PIN_PARG_END()
52                                  );
53
54                         bool detectionVMs = (retVal !=0 ? true:
                               false);
55                         if(detectionVMs) {
56 #ifdef debug
57                                  if ((lpBuffer->AllocationProtect
                                       == 0x40) && (unsigned long)(*
                                       lpBuffer).BaseAddress < 0
                                       x70000000) {
58                                       char textToPrint[4096];
59                                       //sprintf(textToPrint, "[
                                            Orginal] 0x%x\n",
                                            reinterpret_cast<
                                            unsigned long*>(
                                            baseAddress));
60                                       sprintf(textToPrint, "[
                                            ANTES] 0x%x\n", (*
                                            lpBuffer).BaseAddress)
                                            ;
61                                       printMessage(TEXT(
                                            textToPrint));
62
63                                       sprintf(textToPrint, "[
                                            Orginal] 0x%x\n",
                                            getPageContent(
                                            reinterpret_cast<
                                            unsigned long*>((*
                                            lpBuffer).BaseAddress)
                                            ));
64                                       printMessage(TEXT(
                                            textToPrint));
65
66                                       sprintf(textToPrint, "[
                                            Orginal] e igual a 0
                                            xfeedbeaf ? %d\n",
                                            getPageContent(
                                            reinterpret_cast<
                                            unsigned long*>((*
                                            lpBuffer).BaseAddress)
                                            ) == 0xfeedbeaf ? true
                                            : false);
67                                       printMessage(TEXT(
                                            textToPrint));
68
69                                       sprintf(textToPrint, "[
                                            DEPOIS]\n");
70                                       printMessage(TEXT(
```

```
                                                        textToPrint));
71                                                  //return retVal;
72                                                  //return 0;
73                                          }
74 #endif
75                                          // 0x40 = PAGE_EXECUTE_READWRITE
                                                // https://docs.microsoft.com/
                                                en-us/windows/desktop/memory/
                                                memory-protection-constants
76                                          // baseAddress > 0x70000000 is
                                                OS reserved
77                                          if ((lpBuffer->AllocationProtect
                                                == 0x40) && ((unsigned long)
                                                baseAddress < 0x70000000)) {
78                                                  if (*baseAddress == 0
                                                        xfeedbeaf) {
79                                                          return 0;
80                                                  }
81                                          }
82                                  }
83                                  return retVal;
84          }
85
86          static unsigned long getPageContent(unsigned long *p) {
87                  __try {
88                          //printf("%x\n", *p);
89                          return *p;
90                  }
91                  __except (EXCEPTION_EXECUTE_HANDLER) {
92                          char textToPrint[4096];
93                          WINDOWS::DWORD dw = WINDOWS::GetLastError
                                ();
94                          sprintf(textToPrint, "[GetLastError] %d\n
                                ", dw);
95                          printMessage(TEXT(textToPrint));
96                          return NULL;
97                  }
98          }
99
100 protected:
101         PROTO GetPrototypeFunction(char *funcName) {
102                 return PROTO_Allocate(
103                         PIN_PARG(WINDOWS::SIZE_T),
104                         CALLINGSTD_STDCALL,
105                         funcName,
106                         PIN_PARG(WINDOWS::LPCVOID),
107                         PIN_PARG(WINDOWS::
                                PMEMORY_BASIC_INFORMATION),
108                         PIN_PARG(WINDOWS::SIZE_T),
```

```
109                         PIN_PARG_END()
110                         );
111         }
112 
113         void ReplaceFunctionSignature(RTN rtn, PROTO proto, bool
               isUnicode) {
114                 printMessage("\t-> [TLS_GV]
                       ReplaceFunctionSignature\n");
115 
116                 RTN_ReplaceSignature(rtn, (AFUNPTR)
                       WrapperPageQueryFeedbeaf::myPageQuery,
117                         IARG_PROTOTYPE, proto,
118                         IARG_ORIG_FUNCPTR,
119                         IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
120                         IARG_FUNCARG_ENTRYPOINT_VALUE, 1,
121                         IARG_FUNCARG_ENTRYPOINT_VALUE, 2,
122                         IARG_CONTEXT,
123                         IARG_BOOL, isUnicode,
124                         IARG_END
125                         );
126         }
127 
128 
129         virtual RTN FindRTNByNameA(IMG img) {
130                 char *tmp = _strdup(this -> funcName);
131                 // RTN rtn = RTN_FindByName(img, strcat(tmp, "A")
                       );
132                 RTN rtn = RTN_FindByName(img, tmp);
133                 free(tmp);
134                 return rtn;
135         };
136 
137 public:
138         WrapperPageQueryFeedbeaf():PinWrapperWinAPI() {
139                 strcpy(this -> funcName, "VirtualQuery");
140         }
141         ~WrapperPageQueryFeedbeaf(){};
142 };
```

B.3 Detecção por Thread Local Storage (TLS)

```
1 /*
2  * @filename              WrapperGetUserName.h
3  * @version                       1.0, December 2018
4  * @author                        A. da Silva
5  * @description
6  *
7  * This program is free software: you can redistribute it and/or
      modify
```

```
8   * it under the terms of the GNU General Public License as
       published by
9   * the Free Software Foundation, either version 3 of the License,
        or
10  * any later version.
11  *
12  * This program is distributed in the hope that it will be useful
       ,
13  * but WITHOUT ANY WARRANTY; without even the implied warranty of
14  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15  * GNU General Public License for more details.
16  *
17  * You should have received a copy of the GNU General Public
       License
18  * along with this program.  If not, see <http://www.gnu.org/
       licenses/>.
19  */
20
21 #include "PinWrapperWinAPI.h"
22
23 // This value may have to be adjusted.
24 #define PIN_SLOTS 6
25
26 int count = 0;
27
28 class WrapperTLSGetValue: public PinWrapperWinAPI
29 {
30         private:
31 /**
32  Wrapper of TlsGetValue Windows APIs. Spoofs the return if needed
33 before returning to the caller
34  @param orig a pointer to the address of the original function (
       GetModuleHandleA/GetModuleHandleW)
35  @param dwTlsIndex The TLS index that was allocated by the
       TlsAlloc function.
36  @param ctx a pointer to a CONTEXT Pin structure that stores the
       architectural state of the processor
37  @return a bool indicating whether the execution was successful
38 For more information, check https://msdn.microsoft.com/pt-br/
      library/windows/desktop/ms686812(v=vs.85).aspx
39  */
40 static WINDOWS::LPVOID myTlsGetValue(AFUNPTR orig,
41    WINDOWS::DWORD dwTlsIndex,
42    CONTEXT *ctx,
43    bool isUnicode)
44 {
45         WINDOWS::LPVOID retVal = 0;
46         // TlsGetValue e muito executado, logo os prints
              desnecassarios para o rastreio do
```

```
47          // funcionamento serao omitidos
48          // printMessage("\t-> [TLS_GV] Executando\n");
49          PIN_CallApplicationFunction(ctx, PIN_ThreadId(),
50          CALLINGSTD_STDCALL, orig, NULL,
51          PIN_PARG(WINDOWS::LPVOID), &retVal,
52          PIN_PARG(WINDOWS::DWORD), dwTlsIndex,
53          PIN_PARG_END()
54          );
55
56          // bool detectionVMs = checkDetection(ctx, "GetUserName",
                 (char *)lpBuffer, isUnicode);
57          bool detectionVMs;
58
59          // Checa se houve um valor de retorno
60          // detectionVMs = true;
61          detectionVMs = (retVal !=0 ? true: false);
62
63          if(detectionVMs)
64          {
65  char textToPrint[4096];
66  // Pin utiliza por padrao os 5 primeiros slots TLS
67  // Os valores previamente existentes sao deslocados em 5
68  // O que estava no slot 1, vai para o slot 6
69  // Logo, so precisamos spoofar os 5 primeiros slots
70   if (dwTlsIndex < PIN_SLOTS)
71   {
72           printMessage("\t-> [TLS_GV] Asked for TLS values,
                 spoofing return value\n");
73           sprintf(textToPrint, "[Orginal] %d\n", reinterpret_cast<
                 int>(retVal));
74           printMessage(TEXT(textToPrint));
75
76           retVal = (WINDOWS::LPVOID) 0;
77
78          sprintf(textToPrint, "[Modificada] %d\n\n",
               reinterpret_cast<int>(retVal));
79          printMessage(TEXT(textToPrint));
80  }
81          }
82          return retVal;
83  }
84          protected:
85  PROTO GetPrototypeFunction(char *funcName)
86  {
87          return PROTO_Allocate(
88          PIN_PARG(WINDOWS::LPVOID),
89          CALLINGSTD_STDCALL, funcName,
90          PIN_PARG(WINDOWS::DWORD),
91          PIN_PARG_END()
```

```
92          );
93  }
94
95  void ReplaceFunctionSignature(RTN rtn, PROTO proto, bool
        isUnicode)
96  {
97          printMessage("\t-> [TLS_GV] ReplaceFunctionSignature\n");
98
99          RTN_ReplaceSignature(rtn, (AFUNPTR) WrapperTLSGetValue::
                myTlsGetValue,
100         IARG_PROTOTYPE, proto,
101         IARG_ORIG_FUNCPTR,
102         IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
103         IARG_CONTEXT,
104         IARG_BOOL, isUnicode,
105         IARG_END
106         );
107 }
108
109 // No caso do TlsGetValue nao ha uma versao para ANSI e para
        UNICODE
110 // Logo, nao ha uma versao A e W
111 virtual RTN FindRTNByNameA(IMG img)
112 {
113         char *tmp = _strdup(this -> funcName);
114         // RTN rtn = RTN_FindByName(img, strcat(tmp, "A"));
115         RTN rtn = RTN_FindByName(img, tmp);
116         free(tmp);
117         return rtn;
118         };
119
120         public:
121 WrapperTLSGetValue():PinWrapperWinAPI()
122 {
123         strcpy(this -> funcName, "TlsGetValue");
124 }
125 ~WrapperTLSGetValue(){};
126 };
```