Reduzindo a Superfície de Ataque dos Frameworks de Instrumentação Binária Dinâmica

Ailton da Silva dos Santos Filho[1], **Eduardo Luzeiro Feitosa**[1]

[1]Instituto de Computação (IComp) - Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brazil

{assf,efeitosa}@ufam.edu.br

Abstract. *This article proposes countermeasures to mitigate anti-instrumentation techniques, especially those that exploit the increase in attack surface produced by dynamic binary instrumentation tools, allowing for attacks such as arbitrary code execution . Proofs of concept were developed and tested in common controlled environment set of anti-instrumentation techniques. As a result, it is argued that it is possible to reduce the exploitable attack surface of DBI tools by mitigating anti-instrumentation techniques.*

Resumo. *Este artigo propõe contramedidas para mitigar técnicas de anti-instrumentação, em especial as que exploram o aumento da superfície de ataque produzido por ferramentas de instrumentação binária dinâmica, permitindo ataques como a execução de código arbitrário. Provas de conceito foram desenvolvidas e testadas em ambiente controlado com um conjunto de técnicas anti-instrumentação. Como resultado, é apresentado que é possível reduzir a superfície de ataque explorável das ferramentas DBI através da mitigação das técnicas anti-instrumentação.*

1. Introdução

A Instrumentação Binária Dinâmica (*Dynamic Binary Instrumentation*), ou simplesmente DBI, vem sendo empregada em diversas soluções de segurança, como *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] e análise de *malwares* [Kulakov, 2017]. Por permitir a análise em tempo de execução de aplicações, a nível binário, através da inserção de instruções arbitrárias (de análise) no código da aplicação, a instrumentação binária dinâmica auxília na análise de segurança do comportamento de aplicações, inspecionando estruturas como registradores e a pilha do processo, dentre outras capacidades.

No entanto, à medida que soluções e ferramentas baseadas em DBI ganham popularidade, os desenvolvedores de aplicações maliciosas passaram a incorporar em seus códigos artefatos especialmente construídos para detectar ambientes de análise e ferramentas DBI, a fim de evitarem a detecção de suas aplicações. Polino et al. (2017) realizaram a avaliação de 7.006 amostras de *malwares* e observaram que 15.6% das aplicações utilizavam ao menos uma técnica para evadir análises realizadas por ferramentas DBI. Além disso, recentemente alguns autores ([Zhechev, 2018] e [Sun et al., 2016]) vêm questionando a aplicabilidade das ferramentas DBI na análise de *malwares*. Esses autores afirmam que através de determinadas técnicas anti-DBI é possível não só detectar

as ferramentas DBI (comprometimento da transparência), mas também comprometer os ambientes de análise através da exploração da **superfície de ataque** inserida pelas ferramentas DBI (comprometimento do isolamento).

É nesse contexto que este artigo se insere, ao investigar o comprometimento, por técnicas de evasão anti-instrumentação, dos ambientes de análise que utilizam instrumentação binária dinâmica. O objetivo é desenvolver métodos de proteção contra a fuga da instrumentação binária dinâmica, 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. De maneira direta, foram revisadas as técnicas anti-instrumentação presentes na literatura até o momento, a fim de identificar aquelas que comprometem o isolamento das ferramentas DBI, de acordo com a caracterização de comprometimento de isolamento apresentada por Zhechev (2018) e Sun et al. (2016). Como resultado, detectamos três (03) técnicas de evasão anti-instrumentação que não possuem contramedidas: *Detecção por Thread Local Storage (TLS)*, *Detecção de Cache de Código* e *Negligenciamento do No-eXecute Bit(nx)*). Por fim, apresentamos contramedidas que podem ser adotadas para proteger os ambientes de análise que empregam *frameworks* DBI dessas técnicas anti-instrumentação.

Para tanto, foi empregada uma ferramenta capaz de detectar e mitigar comportamentos evasivos de aplicações maliciosas, conhecida como *PinVMShield* [Rodriguez et al., 2016]. Uma vez que essa ferramenta foi desenvolvida com uma arquitetura de *plugins*, foi possível extendermos as funcionalidades da ferramenta com as nossas contramedidas. Demonstramos a eficácia das contramedidas desenvolvidas através de testes realizados com provas de conceito (PoC) das técnicas de evasão anti-instrumentação presentes na literatura, quando disponíveis online, quando não, desenvolvemos PoCs das técnicas anti-instrumentação. Adicionalmente, computamos os impactos no desempenho computacional inseridos pelo uso das contramedidas apresentadas, através da ferramenta *SPEC CPU 2006* - largamente utilizada na literatura.

Dentre as contribuições, destacam-se: (i) o desenvolvimento de provas de conceito (PoC) das técnicas de evasão que podem levar à fuga da instrumentação binária dinâmica e não possuem contramedidas; (ii) o 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; e (iii) a demonstração da capacidade do *framework* DBI Pin de manter seu isolamento e integridade quando garantida a proteção das suas estruturas críticas, apesar da transparência não ter sido assegurada em todos os casos.

2. Instrumentação Binária Dinâmica

Nethercote (2004) define a instrumentação binária dinâmica 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. Dentres suas vantagens frente a outras técnicas de análise, o autor destaca que ela: (i) não requer qualquer preparação prévia para 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) não requer acesso ao código fonte da aplicação cliente ou recompilação da mesma. Rodriguez et al. (2016) acrescentam ainda: (i) ser independente de linguagem de programação e compilador utilizados para a geração da aplicação cliente; e (ii) possuir controle absoluto sobre a execução da aplicação a ser analisada.

No entanto, a instrumentação binária dinâmica também possui desvantagens. Nethercote (2004) aponta que o custo computacional da instrumentação pode afetar a execução das aplicações em análise. Também afirma que existe um aumento na dificuldade de implementação, uma vez que é uma tarefa árdua para o desenvolvedor 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, 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.

2.1. Exploração dos Frameworks DBI

Diversas técnicas de evasão de *frameworks* DBI baseiam-se na busca por artefatos no espaço de memória do processo sob análise, devido as mudanças inseridas por tais ferramentas. Assim, para garantir a compreensão das técnicas anti-evasão, esta subseção apresenta os conceitos relacionados à estrutura e ao funcionamento da memória de um processo, especificamente quando um framework DBI está presente. Observando que está explicação considera um sistema operacional **Windows**, principal alvo das técnicas anti-instrumentação introduzidas pela literatura [Falcón, 2012; Rodriguez et al., 2016; Sun et al., 2016; Hron e Jermář, 2014].

A Microsoft (2017) 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 1 ilustra, a esquerda, como é estruturado o espaço de endereços virtuais no sistema operacional Windows em uma arquitetura 64 bits e, a direita, como é estruturado o espaço de endereços virtuais quando a aplicação está sob análise de um *framework* DBI.

Nota-se, em ambas estruturas da Figura 1, 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. Cada região em cinza, chamada de segmento de memória, pode ser composta por apenas uma página[1] de memória ou por um conjunto delas. Contudo, na parte direira da Figura 1, nota-se a presença dos Caches de Código, partes essenciais para o funcionamento de soluções DBI, pois é neles que residem o código da aplicação instrumentada e informações de controle específicas 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 na ferramenta DBI 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*).

[1]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).

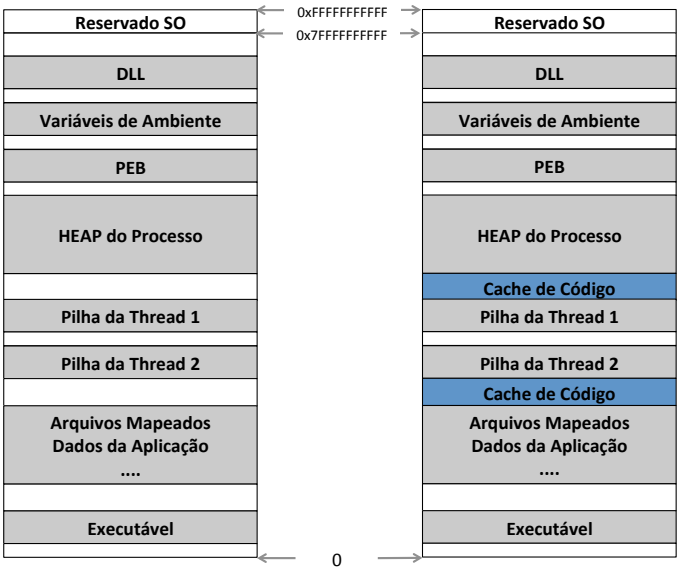

Figura 1. Estrutura da memória de um processo. Adaptado de [Ligh et al.**, 2014].**

Alguns artefatos inseridos pelos *frameworks* DBI aumentam a superfície de ataque, permitindo não apenas a detecção da ferramenta DBI, mas também a fuga e a alteração (subversão) no fluxo de execução do processo de instrumentação. Como consequência, 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*). Além disso, a execução de códigos arbitrários, mesmo em ambientes controlados, pode levar a comportamentos maliciosos pelo ambiente de análise, infecção do sistema por *malwares* e até o comprometimento do computador executando as análises DBI [Carpenter et al., 2007].

2.2. Técnicas para Fuga da Instrumentação

As principais técnicas para alcançar a fuga da instrumentação e subversão do fluxo de execução são baseadas na manipulação de estruturas críticas do *framework* DBI, como Cache de Código, pilha e *callbacks* aplicação-*framework*. As técnicas Detecção de *Thread Local Storage*, Detecção de *Cache* de Código ameaçam o isolamento dos *frameworks* DBI porque expõem endereços de *Caches* de Código para a aplicação sob análise, enquanto a técnica Negligenciamento do *No-eXecute Bit* permite a execução de dados como código, possibilitando diversos ataques, como *buffer overflow*.

2.2.1. Detecção de Thread Local Storage (TLS)

Segundo a Microsoft (2018b), *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.

Frameworks DBI fazem uso intensivo de 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 (2018b), no Windows toda a interação com o TLS é intermediada pelo SO através das APIs: *TlsAlloc*, *TlsSetValue*, *TlsGetValue* e *TlsFree*. O *framework* Pin faz uso do TLS, o que permite a uma aplicação sob análise fazer uma verificação dos índices TLS sendo utilizados, revelando a presença do *framework* DBI.

A Figura 2 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.

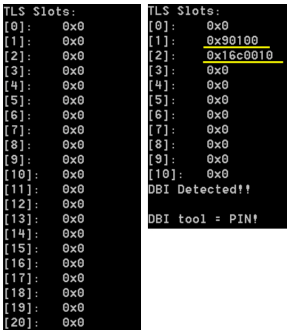

Figura 2. Comparação entre a estrutura TLS com e sem o Pin [Sun et al.**, 2016].**

2.2.2. Detecção de Cache de Código

Dentre os mecanismos exploráveis que podem levar à 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 (fuga da instrumentação). Como forma de ilustrar uma dessas técnicas e deixar o leitor mais familiarizado com o tema, a Figura 3 apresenta o conteúdo de um Cache de Código (Figura 3.b) para o *framework* DBI Pin, em relação ao código original da aplicação sem instrumentação (Figura 3.a).

Na Figura 3.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 possa determinar o endereço do Cache de Código e relacioná-lo 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 instruções de controle do Pin na Cache de Código, reescrever o código da aplicação com instruções de desvio, redire-

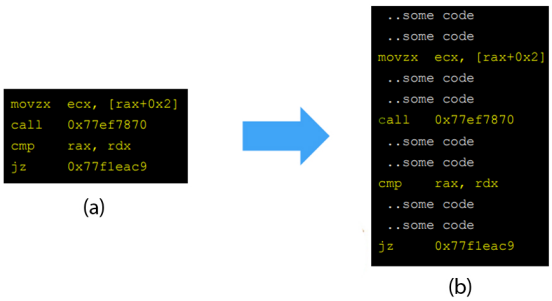

Figura 3. Ilustração do conteúdo de um Cache de Código em relação ao código original [Intel Corporation, 2013].

cionando 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* DBI.

2.2.3. 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 SO. 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.

2.3. Histórico da Evasão dos *Frameworks* DBI

Esta subseção apresenta, de maneira cronológica, os trabalhos relacionados às técnicas de detecção e evasão de ferramentas DBI, abordando 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 (2012) introduziram o primeiro conjunto de técnicas evasivas com o objetivo de detectar o *framework* de instrumentação Pin no Windows. Os autores apresentaram treze técnicas, com provas de conceito (PoCs), 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* DBI, o DynamoRIO, poderia ser detectado pela aplicação em análise, tanto em ambientes Windows quanto Linux. Similarmente, Hron e Jermář (2014) propuseram seis 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 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 *Arancino*. Eles também validaram a eficácia dessas contramedidas através da ferramenta *eXait* [Falcón, 2012].

Recentemente, Zhechev (2018) publicou uma dissertação de mestrado focada no questionamento se *frameworks* DBI são ferramentas apropriadas para analisar programas desconhecidos e/ou potencialmente maliciosos porque não são capazes de garantir as características isolamento e transparência, e por levarem ao aumento da superfície de ataque. Além de demonstrá-la, o autor investigou as razões que tornam possivel a fuga da instrumentação. Por fim, o autor apresentou treze novas técnicas de evasão contra o *framework* DBI Pin em ambientes Linux.

3. Novas Contramedidas

Esta seção 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).

3.1. *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*, o que permite estendê-la para cobrir diferentes técnicas evasivas. É distribuída sob a licença GNU GPL versão 3 e possui o código fonte disponível online [2].

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

[2] `https://bitbucket.org/rjrodriguez/pinvmshield/`

discutidos em detalhes nas subseções subsequentes são classes que herdam os atributos e métodos da classe abstrata *PinWrapperWinAPI*, o que permite que seja feito com facilidade a interceptação de chamadas a APIs do Windows, através do método *ReplaceWinAPI*.

3.2. Contramedidas

Esta subseção descreve as novas abordagens de mitigação para técnicas anti-instrumentação que podem levar a fuga da instrumentação e que não possuem contramedidas 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. As provas de conceito desenvolvidas estão disponíveis online [3].

3.2.1. Contramedida para detecção por *Thread Local Storage* (TLS)

Uma vez que a técnica de detecção por *Thread Local Storage* (TLS) consiste na verificação em tempo de execução das posições do TLS sendo utilizadas, propõe-se para mitigá-la uma abordagem baseada em realizar o controle das chamadas às APIs relacionadas ao TLS e utilizar uma política de gerenciamento das posições TLS retornadas por essas APIs.

Para realizar o controle das chamadas às APIs relacionadas ao TLS foram utilizadas as funcionalidades *ReplaceFunctionSignature* do *PinVMShield*, providas 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. Assim, toda vez que uma das APIs sendo monitoradas (*TlsAlloc*, *TlsSetValue*, *TlsGetValue*, *TlsFree*) é invocada, ocorre a interceptação da chamada, a aplicação da política de gerenciamento de posições do TLS e a realização da chamada de sistema modificada.

A política de gerenciamento das 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 ($N = 2$), por exemplo, e a aplicação cliente faz uma requisição a uma dessas posições, como à posição 0, a chamada de sistema é modificada para alvejar a posição deslocada em N, ou seja, a posição 2 que não está sendo utilizada pela ferramenta DBI. 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.

Dessa forma, o isolamento entre as posições do TLS alocadas e as utilizadas pelo

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

framework DBI é garantido logicamente, ou seja, não há a separação real. Tanto a ferramenta DBI, quanto a aplicação cliente continuam compartilhando a mesma estrutura de dados. No entanto, através do monitoramento ativo das interações com o TLS e da aplicação da política de controle de posições foi possível impedir interações da aplicação cliente com o *framework* DBI, através do TLS.

Essa abordagem conta como principal desvantagem a redução efetiva do número total de posições do TLS disponíveis para a aplicação cliente, 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 da técnica anti-instrumentação - seria possível detectar esta contramedida, através da contagem das posições disponíveis do *Thread Local Storage*.

3.2.2. Contramedida para Detecção de Cache de Código

Para mitigar a técnica anti-instrumentação que detecta Caches de Código, a contramedida proposta estabelece uma política de identificação das regiões de memória virtual e controle de acesso às regiões de memória que são Caches de Código, definindo-as como *endereços de memória protegidos*. 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. Na prova de conceito proposta foi adotada a política de *blacklist* para estabelecer a identificação das regiões de memória virtual que a aplicação pode interagir, uma vez que o framework DBI Pin fornece nativamente APIs que facilitam a monitoração dos Caches de Código, como a *CODECACHE_AddTraceInsertedFunction* que permite que uma função seja notificada cada vez que um Cache de Código é criado. Assim, na contramedida proposta, uma lista contendo as faixas de endereço utilizadas como Caches de Código pela ferramenta DBI é preenchida em tempo de execução pelo *PinVMShield* através da API do Pin citada. Na prova de conceito desenvolvida, a função notificada pela criação de um Cache de Código é a função que faz a gerência da lista de regiões de memória protegidas.

Ademais, a política de *blacklist* foi adotada, em detrimento a de *whitelist*, devido a segunda tender a ser 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*.

Para realizar o controle de acesso à 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 faz referência a um endereço de um Cache de Código, é retornado o valor zero, indicando que a chamada de sistema falhou [Microsoft,

2018a].

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.

3.2.3. Contramedida para o negligenciamento do *No-eXecute Bit* (nx)

A contramedida proposta consiste em realizar uma implementação, semelhante a dos SOs, de verificação do bit de execução antes de qualquer código da aplicação ser executado, uma vez que esse comportamento está ausente nas ferramentas DBI. Nos sistemas Windows, esse mecanismo de verificação é conhecido como *Data Execution Prevention* (DEP) ou Prevenção de Execução de Dados [Microsoft, 2019]. Nos *frameworks* DBI, imediatamente antes do compilador *just-in-time* do *framework* ser invocado para construir os Caches de Código, deveria ser verificado se a região de memória de onde é proveniente o código sendo utilizado como entrada possui permissões de execução. Em caso negativo, da mesma forma que o sistema operacional faz, deveria ser lançada uma exceção com status *STATUS_ACCESS_VIOLATION*.

A implementação desta contramedida consiste em construir lista de Caches de Códigos através do mesmo mecanismo utilizado para construir a *blacklist*, apresentado na subseção 3.2.2, 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 sistema opercaional *VirtualQuery*. Quando é determinado que existe a permissão de execução, a ferramenta DBI continua sua execução normal. Caso contrário, a execução do programa é suspensa temporariamente. Assim, o analista utilizando o *framework* DBI é notificado através de um *log* textual (arquivo de texto), podendo tomar uma ação como por exemplo: 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 SO.

A seção seguinte discute os impactos de desempenho causados pelo uso das contramedidas apresentadas nas aplicações sendo analisadas. Além disso, são discutidos como as contramedidas propostas poderiam ser detectadas ou evadidas por um desenvolvedor de *malwares*.

4. Desempenho da Contramedidas Propostas

Para determinar a eficácia das contramedidas propostas de maneira prática foram implementadas provas de conceito (PoC) funcionais. Esses artefatos encontram-se disponíveis online [4] sob uma licença BSD de uso. Como já mencionado anteriormente, as provas de conceito foram incorporadas à ferramenta *PinVMShield* [Rodriguez et al., 2016] para a validação de suas eficácias.

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

O cenário de testes consistiu 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. A escolha pelo Windows 7 se deve ao mesmo ter sido adotado como SO de quase todas as propostas existentes na literatura e da ferramenta *PinVMShield*. O SPEC CPU2006 [CPU2006, 2006] é 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., 2005; Arafa, 2017; Bruening et al., 2012].

4.1. Mitigação e *Arms Race*

A técnicas de detecção por TLS e 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, permitindo que o analista tomasse 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* 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 de detecção 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 atacante 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 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.

4.2. Desempenho

Para o teste de desempenho, foi utilizado o conjunto de componentes de testes *SPECint2006* - assim como desenvolvedores do Pin [Luk et al., 2005]-, que possui um conjunto de 12 aplicações de testes diferentes utilizadas para se obter métricas de desempenho. Ressalta-se, no entanto, que apenas 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 artigo.

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 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 1 sumariza os resultados obtidos para cada uma das aplicações de teste do *SPECint2006*.

Tabela 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

A primeira coluna lista os nomes das aplicações de *benchmark* do *SPEC*. A segunda coluna informa o tempo (em segundos) da execução do 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 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. Segundo os desenvolvedores do Pin [Luk et al., 2005], o *overhead* imposto por uma *pintool* pode variar de acordo com a aplicação sendo instrumentada, uma vez que o *framework* DBI opera as instruções de tal aplicação, o que pode ser observado na Tabela 1. Diversos fatores influenciam nessa variação, como: as quantidades de desvios indiretos, de reuso de código e de registradores sendo utilizados.

4.3. Discussão

As condições de *arms race* podem levar um *malware* evasivo a detectar o *framework* DBI, 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 *framework* DBI através da mitigação das técnicas evasivas que exploram as vulnerabilidades desses *frameworks*, evitando ataques de execução de código arbitrário. Assim, este trabalho resulta em uma resolução que difere da apresentada por Zhechev (2018), concluindo que os *frameworks* DBI não tornam necessariamente os ambientes que os utilizam menos seguros e podem ser utilizados para fins 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.

5. Considerações Finais

Este artigo focou em apresentar novas contramedidas para técnicas que afetam a superfície de ataque explorável introduzida pelas ferramentas DBI. De forma prática, foi demonstrado que os *frameworks* DBI 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 devidamente equipados com as ferramentas apropriadas (como contramedidas anti-instrumentação e anti-evasão). Para isso, foram propostas e desenvolvidas contramedidas (PoCs) sobre a ferramenta PinVMShield, uma solução desenvolvida para mitigar técnicas de evasão diversas. Após os testes de eficácia, foi realizada uma análise do *overhead* temporal introduzido pelas contramedidas desenvolvidas (aproximadamente 60% em média).

Apesar dos esforços dos desenvolvedores de ferramentas de segurança, existem técnicas anti-instrumentação que ainda 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;
- 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 -, análises e pesquisas sobre se os *frameworks* DBI são a opção ótima para a análise de software não seguro e/ou potencialmente malicioso ainda são um ponto em aberto;

6. Agradecimentos

Este trabalho foi desenvolvido com o apoio do Governo do Estado do Amazonas por meio Fundação de Amparo à Pesquisa do Estado do Amazonas, com a concessão de bolsa de estudo.

Referências

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

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

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

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

Deng, Z., Zhang, X. e Xu, D. (2013). SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization. Annual Computer Security Applications Conference *1*, 289–298.

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

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

Hron, M. e Jermář, J. (2014). SafeMachine malware needs love, too.

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

Kulakov, Y. (2017). MazeWalker - Enriching Static Malware Analysis. [Online; `https://bit.ly/2Mzodir`]. Acessado em 18.12.2018.

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

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. e Hazelwood, K. (2005). 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 pp. 190–200, ACM, New York, NY, USA.

Mariani, S., Fontana, L. e Gritti, F. (2016). PinDemonium: a DBI-based generic unpacker for Windows executables. [Online; `https://ubm.io/30WYsvy`]. Acessado em 18.12.2018.

Microsoft (2017). Virtual Address Space (Windows). [Online; `https://bit.ly/2W7p890`].

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

Microsoft (2018b). Thread Local Storage. [Online; `https://bit.ly/2GeBJTw`].

Microsoft (2019). 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://bit.ly/2XqxRVa`].

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

Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontata, L., Gritti, F. e 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).

Rodriguez, R. J., Gaston, I. R. e Alonso, J. (2016). Towards the Detection of Isolation-Aware Malware. IEEE Latin America Transactions *14*, 1024–1036.

Stamatogiannakis, M., Groth, P. e Bos, H. (2015). Looking Inside the Black-Box: Capturing Data Provenance Using Dynamic Instrumentation. In Provenance and Annotation of Data and Processes, (Ludäscher, B. e Plale, B., eds), pp. 155–167, Springer International Publishing, Cham.

Sun, K., Li, X. e 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.

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