Mitigando Técnicas de Anti-Instrumentação em DBI: Contramedidas baseadas em *Overhead* e Transparência

Francisco S. S. Neto[1], Henrique B. Campelo[1], Euler V. Silva[1,2], Eduardo L. Feitosa[1]

[1]Instituto de Computação – (ICOMP) – Universidade Federal do Amazonas (UFAM)
CEP – 69.077-000 – Manaus – AM – Brasil,

[2]Instituto Federal de Educação, Ciência e Tecnologia do Amazonas (IFAM)
CEP 69.020-120 – Manaus – AM – Brasil

{francisconeto,henrique.borges,eulervieira,
efeitosa}@icomp.ufam.edu.br

Abstract. *In this paper, we introduce three new countermeasures to mitigate overhead and transparency based anti-instrumentation techniques employed by context-aware malware to detect the presence of Dynamic Binary Instrumentation (DBI). We validated these countermeasures through proofs-of-concept in a controlled environment. The results indicate that it is possible to reduce the attack surface of such malware, promoting greater transparency and resilience in DBI-instrumented environments.*

Resumo. *Apresentamos neste artigo três novas contramedidas para mitigar técnicas de anti-instrumentação baseadas em overhead e transparência, empregadas por malwares cientes de contexto para detectar a presença de instrumentação binária dinâmica (DBI). Validamos as contramedidas por meio de provas de conceito em ambiente controlado. Os resultados indicam que é possível reduzir a superfície de ataques desses malwares, promovendo maior transparência e resiliência em ambientes instrumentados por DBI.*

1. Introdução

A análise de *malwares* de forma dinâmica tem se consolidado como um componente essencial na área de cibersegurança, exigindo soluções cada vez mais sofisticadas de detecção e neutralização. Dentre as estratégias de análise dinâmica, a Instrumentação Binária Dinâmica (*Dynamic Binary Instrumentation* ou simplesmente DBI) destaca-se por permitir a inserção de instruções adicionais, em tempo de execução, sem exigir acesso ao código-fonte nem recompilação. Contudo, o uso de DBI também instigou o surgimento de técnicas evasivas específicas com o objetivo de detectar e contornar ambientes instrumentados por DBI. Essas abordagens, conhecidas como técnicas de evasão anti-instrumentação, exploram vulnerabilidades estruturais dos frameworks DBI, especialmente o *overhead* introduzido no tempo de execução e as limitações em garantir total transparência durante a análise. Por exemplo, a análise realizada por [Polino et al. 2017], em 7.006 amostras de *malware*, identificou que 15,6% delas incorporavam mecanismos capazes de identificar e evitar DBI.

Diante desse desafio, o presente artigo propõe três novas contramedidas voltadas à proteção de ambientes DBI, concentradas em três classes de técnicas evasivas: (i) *Environment Variables*, que detecta nomes de variáveis associados a ferramentas de instrumentação; (ii) *DBI fingerprints*, que permite identificar artefatos característicos de ambientes instrumentados; e (iii) *Execution Time*, que exploram a diferença perceptível entre o tempo real de determinadas operações em ambientes nativos e instrumentados.

2. Instrumentação Binária Dinâmica

Segundo [Nethercote 2004], a Instrumentação Binária Dinâmica (DBI) é uma técnica que não requer qualquer preparação prévia para análise do programa a ser instrumentado, como injeção de bibliotecas ou alterações no código binário do programa; cobre naturalmente todo o código do programa, o que pode não ser possível nas análises estáticas, especialmente quando o programa gera códigos dinamicamente; e não requer acesso ao código fonte do programa ou recompilação do mesmo. [Rodríguez et al. 2016] acrescentam ainda: (i) ser independente de linguagem de programação e compilador utilizados para a geração do programa; e (ii) possuir controle absoluto sobre a execução do programa a ser analisado.

Apesar das vantagens, DBI apresenta duas limitações recorrentes. A primeira refere-se ao *overhead* gerado pela inserção de código auxiliar, que pode introduzir atrasos na execução do programa analisado. A segunda envolve a exposição de estruturas não naturais no ambiente em tempo de execução, o que pode comprometer a transparência do processo de instrumentação. Essas limitações têm sido alvo de exploração por diferentes técnicas evasivas [Polino et al. 2017, Rodríguez et al. 2016, Santos Filho and Feitosa 2019].

3. Técnicas de Anti-Instrumentação para DBI

Embora a literatura acadêmica sobre anti-instrumentação em DBI apresenta diferentes ideias, neste estudo focamos em três técnicas.

Environment Variables: exploram estruturas acessíveis durante a execução de um processo que armazenam parâmetros de configuração definidos por sistemas operacionais, aplicações ou DBI. O Intel Pin, por exemplo, usa a variável `PIN_INJECTOR64_LD_LIBRARY_PATH` para definir o caminho para bibliotecas compartilhadas utilizadas durante a instrumentação. Assim, sua presença pode ser interpretada pelo binário em análise como um forte indicativo da presença de instrumentação.

DBI fingerprints: exploram padrões de código característicos, como cadeias estáticas de caracteres (`pin.exe`, `pinvm.dll` ou prefixos `PIN_`, por exemplo) presentes na memória do processo e que, consequentemente, podem indicar a atuação de um DBI.

Execution Time: exploram as diferenças perceptíveis entre a duração real de determinadas operações em ambientes nativos e instrumentados. DBI costuma introduzir sobrecarga adicional durante a execução de chamadas comuns e instruções simples, resultando em atrasos que podem ser detectados por meio de medições temporais precisas.

4. Concepção das Contramedidas

Analisar contramedidas contra técnicas de evasão anti-DBI exigem a criação de ambientes controlados. Por isso, elaboramos três aplicações de teste, uma para cada técnica de

evasão. Além disso, também propomos uma ferramenta (*pintool*), chamada TOMWare, para detecção das técnicas de evasão e execução das contramedidas propostas.

A **primeira aplicação** procura por strings definidas (`PIN_CRT_TZDATA`, `PIN_APP_LD_LIBRARY_PATH` e `PIN_VM_LD_LIBRARY_PATH`, especificamente) utilizando a função *GetEnvironmentStringsW*. Em seguida, libera o bloco de variáveis de ambiente e executa outro método, chamando `_dupenv_s`, para recuperar explicitamente o valor de `PIN_CRT_TZDATA`, a variável de ambiente testada. Caso encontre a string, ela informa que o Pin foi detectado.

A **segunda aplicação** invoca a função *VirtualQuery* para visitar, página por página, toda a memória do processo. Em cada região válida, a aplicação realiza uma comparação byte a byte, usando *memcmp*, procurando por `"PIN_"`, `"pin.exe"`, `"pinvm.dll"` ou `"pinipc.dll"`.

A **terceira aplicação** baseia-se na premissa de que, em condições normais, pedir a execução da função *Sleep (50 ms)* devolve um tempo próximo de 50–55 ms. Valores muito acima disso sugerem *overhead* que pode ser vinculado a um instrumentador DBI.

4.1. TOMWare

A ferramenta TOMWare[1] foi desenvolvida sobre o framework Intel Pin, para oferecer uma camada de blindagem capaz de neutralizar mecanismos de detecção direcionados ao Intel Pin, com foco em estratégias baseadas em *overhead* e transparência. A TOMWare é compatível com sistemas Windows 10 (ou superior) na arquitetura x64. Sua compilação pode ser realizada utilizando o Microsoft Visual Studio Community 2022 (64 bits), e a ferramenta é plenamente funcional com a versão 3.28 do Intel Pin. Cada aplicação apresentada nas seções anteriores foi implementada como um módulo independente, permitindo sua ativação de forma seletiva, conforme o cenário de análise.

5. Contramedidas Propostas

As contramedidas propostas visam neutralizar falhas exploradas por aplicações maliciosas, ampliando a capacidade dos frameworks DBI de manter um ambiente de análise dinâmica confiável e resiliente diante de mecanismos de evasão.

5.1. SanitizePinEnvVars

O instrumentador Pin cria variáveis de ambiente com o prefixo `PIN_` e a contramedida ***SanitizePinEnvVars*** realiza a remoção controlada dessas variáveis com o objetivo de eliminar vestígios deixados pelo Intel Pin durante a instrumentação.

Em determinadas situações, ao alterar dados das variáveis de ambiente, o Windows detecta que o bloco de variáveis associado à estrutura PEB está fragmentado ou não possui espaço suficiente para armazenar as alterações. Assim, ele aloca um novo bloco de memória na região de *heap*, copia todas as variáveis ainda válidas para esse novo bloco e atualiza o ponteiro do bloco no PEB para o novo local em memória. O bloco antigo, embora desalocado logicamente, pode permanecer acessível na memória, por não ser imediatamente liberado, e pode ser explorado por varreduras manuais. Consequentemente,

[1] `https://github.com/TOMWare-DBI/TOMWare`

esse comportamento representa um risco, pois os dados anteriormente armazenados, incluindo assinaturas como `PIN_` podem permanecer acessíveis no bloco antigo, mesmo após a exclusão lógica. A contramedida ***SanitizePinEnvVars*** também trata esse caso e aplica uma etapa de limpeza no espaço de memória do bloco antigo, eliminando completamente os vestígios residuais e monitorando alterações no endereço do ponteiro de ambiente antes e depois de cada remoção. Se for detectada uma realocação, o conteúdo do bloco anterior é sobrescrito, impedindo sua recuperação por métodos de análise de memória.

A contramedida verifica seis variáveis sensíveis associadas ao Pin, a fim de identificar: (i) quando a variável a ser removida não está presente no ambiente do processo, ao executar a função *GetEnvironmentVariableW* e receber como retorno o código de erro `ERROR_ENVVAR_NOT_FOUND`; (ii) quando a variável a ser removida foi excluída diretamente no PEB, sem realocação. Nesse caso, como dados residuais podem permanecer na região final, a função *SecureZeroMemory* é executada sobre a área residual; (iii) quando identifica que o Windows executou uma realocação de blocos.

A execução da contramedida ***SanitizePinEnvVars*** ocorre logo após a injeção do agente Pin, mas antes da chamada *main()* da aplicação analisada, garantindo que qualquer acesso posterior ao ambiente observe um contexto "limpo". Ela não altera a lógica do programa-alvo; apenas a cópia de ambiente em espaço-usuário é modificada. Além disso, nenhum *handle* ou permissão adicional é requerido. Também não deixa resíduos, uma vez que o uso da função *SecureZeroMemory* previne a recuperação de assinaturas por leitura bruta de regiões liberadas.

5.2. InstMemcmpMask

Empacotadores (*packers*) e *malwares* executam verificações de memória, em tempo de execução, para decidir se estão sob instrumentação. Basta, apenas, comparar blocos de bytes com assinaturas que denunciam o DBI em uso (`PIN_CRT_TZDATA`, `pin.exe`, `pinvm.dll`, por exemplo). A contramedida ***InstMemcmpMask*** intercepta doze (12) variantes dessas rotinas[2] no Intel Pin . Cada chamada é redirecionada para um *wrapper*, que executa a função original, captura o resultado e, caso o retorno indique igualdade ($ret = 0$) e o segundo operando corresponda a uma das quatro assinaturas sensíveis (`PIN_`, `pin.exe`, `pinvm.dll` e `pinipc.dll`), ela força o retorno para "diferente" (1). Dessa forma, a comparação falha deliberadamente, impedindo que o binário analisado confirme a presença do Pin por validação de strings.

Para garantir ampla cobertura, a implementação intercepta múltiplas variantes das funções de comparação binária, abrangendo tanto aquelas presentes na biblioteca padrão da linguagem C (*ucrt*) quanto as versões exportadas diretamente pela *ntdll.dll*, como *RtlCompareMemory* e *RtlEqualMemory*. A contramedida não modifica nenhum byte da memória original, apenas ajusta o valor de retorno das funções interceptadas, evitando problemas relacionados à proteção de páginas de memória. Como limitações, a contramedida atua apenas sobre funções de comparação explícita, ou seja, chamadas diretas a rotinas como *memcmp* ou *RtlCompareMemory*. Técnicas que realizam varreduras manuais na memória, utilizando ponteiros ou lógica personalizada, não são interceptadas e podem

[2]O conjunto inclui versões *secure*, insensíveis a caixa (*memicmp*) e funções *wide-char*.

contornar essa proteção. Além disso, comparações realizadas por funções não incluídas na tabela de interceptação também permanecem fora do escopo da contramedida.

5.3. SkewMask

Malwares são capazes de identificar pequenas variações no tempo de execução de chamadas, o que denunciam a presença de instrumentação. A contramedida ***SkewMask*** introduz um mecanismo de compensação temporal sobre funções críticas afetadas pelo *overhead*.

A estratégia consiste em interceptar funções de temporização como *Sleep*/*SleepEx* e medir o tempo real decorrido. Qualquer excesso em relação ao tempo solicitado é somado a um contador global, acessado por operações atômicas. Nas chamadas subsequentes a funções como *GetTickCount*, *QueryPerformanceCounter* e *GetSystemTimePreciseAsFileTime*, a contramedida devolve $valororiginal - contador$, mascarando o impacto da instrumentação e restituindo valores compatíveis com execuções nativas. O mesmo contador é utilizado para todas as APIs, garantindo que diferentes fontes de tempo permaneçam coerentes entre si. Com o atraso invisível, verificações temporais internas deixam de diferenciar uma execução sob Pin de uma execução nativa, elevando a furtividade (transparência) do DBI. O custo de processamento introduzido é mínimo, apenas algumas operações aritméticas e atômicas por chamada, sem alterar a lógica nem a estabilidade da aplicação analisada.

6. Resultados

Para avaliar as contramedidas propostas, foram analisados três cenários: (1) aplicação de teste sem o Pin (execução em bare-metal); (2) aplicação de teste sob a TOMWare, sem a contramedida específica ativada; (3) aplicação de teste sob a TOMWare, com a contramedida específica ativada. Os experimentos foram executados em uma máquina com processador Intel Core i7-1165G7, 16 GB de RAM e sistema operacional Windows 11 x64.

6.1. *SanitizePinEnvVars*

A Figura 1 mostra a aplicação teste criada para inspecionar o ambiente do processo, confirmando a presença da variável `PIN_CRT_TZDATA` lida diretamente do bloco de variáveis mantido pelo PEB, e denunciando o Pin por ter encontrado uma variável com o prefixo `PIN_`.

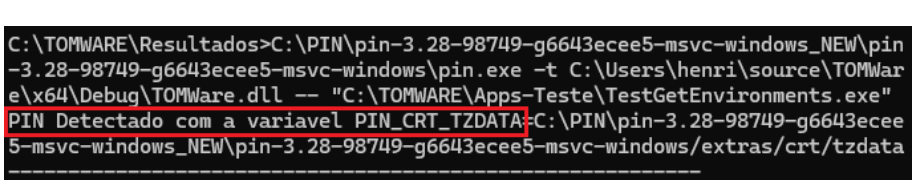

Figura 1. Pin detectado por meio da variável `PIN_CRT_TZDATA`.

A Figura 2 apresenta a execução da mesma aplicação teste sob a TOMWare com a contramedida, que intercepta o programa logo na fase de início, remove a entrada suspeita da lista de ambiente e zera o espaço residual no *heap*, de modo que varreduras subsequentes não encontrem qualquer vestígio da instrumentação.

```
C:\TOMWARE\Resultados>C:\PIN\pin-3.28-98749-g6643ecee5-msv
c-windows_NEW\pin-3.28-98749-g6643ecee5-msvc-windows\pin.e
xe -t C:\Users\henri\source\TOMWare\x64\Debug\TOMWare.dll
-de -- "C:\TOMWARE\Apps-Teste\TestGetEnvironments.exe"
[=] PIN_CRT_TZDATA: removida (in-place)
[=] PIN_APP_LD_LIBRARY_PATH: inexistente
[=] PIN_VM_LD_LIBRARY_PATH: inexistente
[=] PIN_VMLOG: inexistente
[=] PIN_APP_SHORTNAME: inexistente
[=] PIN_LOG: inexistente
Nenhuma variável do PIN detectada.
------------------------------------------------------------
```

Figura 2. Contramedida aplicada: Pin não foi identificado.

A contramedida aplica uma etapa de higienização da memória para evitar que resíduos possam ser recuperados por varreduras, sem interferir na lógica da aplicação ou modificar estruturas internas do sistema operacional.

6.2. *InstMemcmpMask*

A Figura 3 apresenta a saída da aplicação teste de varredura de memória, evidenciando a quantidade expressiva de ocorrências das assinaturas `PIN_`, `pin.exe`, `pinvm.dll` e `pinipc.dll`.

```
Resumo de ocorrências:
  PIN_    : 283
  pin.exe : 12
  pinvm.dll : 15
  pinipc.dll : 19
Alerta: mais de 4 ocorrências de "PIN_" encontradas!
Alerta: mais de 2 ocorrências de "pin.exe" encontradas!
Alerta: mais de 2 ocorrências de "pinvm.dll" encontradas!
Alerta: mais de 2 ocorrências de "pinipc.dll" encontradas!
```

Figura 3. Pin detectado por meio de *Overhead*.

A Figura 4 mostra que, após a ativação da contramedida *InstMemcmpMask*, todas as comparações com essas sequências retornam deliberadamente “diferente”, de modo que o mesmo programa não consegue localizar nenhum dos padrões e, consequentemente, falha em detectar a instrumentação.

```
C:\TOMWARE\Resultados>C:\PIN\pin-3.28-98749-g6643ecee5-msv
c-windows_NEW\pin-3.28-98749-g6643ecee5-msvc-windows\pin.e
xe -t C:\Users\henri\source\TOMWare\x64\Debug\TOMWare.dll
-dm -- "C:\TOMWARE\Apps-Teste\TestMemoryScan.exe"

Ocorrências:

Resumo de ocorrências:
  PIN_    : 0
  pin.exe : 0
  pinvm.dll : 0
  pinipc.dll : 0
```

Figura 4. Contramedida aplicada. Pin não foi identificado.

6.3. *SkewMask*

A contramedida *SkewMask*, por sua vez, introduz um mecanismo de compensação temporal que atua sobre funções como *Sleep*, *QueryPerformanceCounter* e *GetTickCount*. Na Figura 5, a aplicação teste identifica a presença do Pin, pois a medição da função *Sleep()* retorna um valor superior ao esperado.

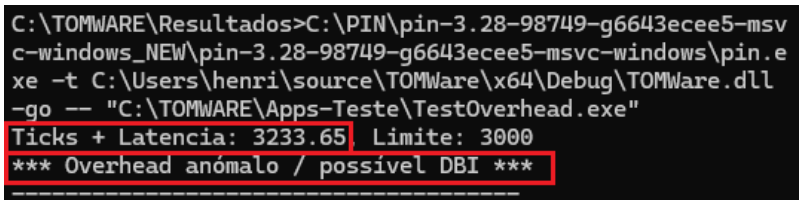

Figura 5. Pin detectado por meio de *Overhead*.

Já na Figura 6, com a contramedida *SkewMask* ativa, os mesmos intervalos retornam a níveis normais, impedindo que o detector reconheça a instrumentação.

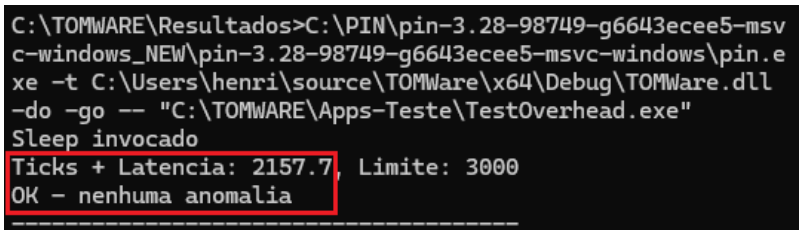

Figura 6. Pin não identificado com a contramedida *SkewMask* aplicada.

7. Considerações Finais

Este artigo apresentou um conjunto de contramedidas voltadas à mitigação de técnicas evasivas que exploram o *overhead* e a transparência de frameworks DBI. As soluções propostas foram concebidas com base na interceptação de funções sensíveis e na manipulação controlada de estruturas do ambiente de execução, sendo implementadas como provas de conceito na ferramenta TOMWare. Todas as contramedidas foram desenvolvidas em espaço de usuário e aplicadas em testes controlados para análise de eficácia e impacto no desempenho.

Apesar dos resultados positivos, persistem desafios importantes no desenvolvimento de contramedidas que preservem a transparência da instrumentação com impacto mínimo. Entre as dificuldades identificadas estão a ausência de soluções genéricas para técnicas de evasão baseadas em temporização e a limitação de mecanismos capazes de ocultar completamente artefatos em memória ou alterações no fluxo de execução introduzidas por instrumentadores.

Os resultados dos experimentos confirmam o dilema clássico: quanto maior a transparência desejada, maior o custo inevitável em tempo de execução. Cada nova contramedida amplia a superfície de interceptação de funções ou instruções e acrescenta processamento extra. A transparência total é, portanto, inalcançável na prática: sempre que se oculta mais um artefato, introduz-se algum grau de latência. A decisão de ativar uma defesa deve considerar o perfil de chamadas da aplicação-alvo; uma técnica inofensiva

para software interativo pode ser proibitiva em uma rotina de alto débito de dados. Da mesma forma, a inclusão de contramedidas adicionais, sobretudo as que operam continuamente, tende a aumentar o *overhead* de forma não linear, exigindo um balanceamento cuidadoso entre invisibilidade e desempenho.

Diante desse cenário, não há solução genérica para todas as formas de evasão por temporização, e a remoção completa de artefatos em memória esbarra na necessidade de preservar a funcionalidade original do programa. Contudo, as contramedidas propostas aumentam significativamente a resiliência do framework, preservando, na medida do possível, o isolamento, a integridade e a previsibilidade do ambiente instrumentado.

7.1. Trabalhos Futuros

Para o futuro, acreditamos que novas estratégias, que combinem monitoramento adaptativo e técnicas reativas, com o objetivo de ampliar a cobertura diante de mecanismos evasivos emergentes serão necessárias. As limitações observadas, como a dificuldade em ocultar artefatos de instrumentação sem comprometer a funcionalidade do binário e os desafios associados a medições de tempo sensíveis, evidenciam a necessidade de soluções dinâmicas ajustadas ao perfil de execução da aplicação.

Assim, a pesquisa irá seguir em três direções prioritárias:

1. Elaborar contramedidas adaptativas, que ativem suas proteções somente diante de padrões suspeitos, com base em heurísticas como uso das funções sensíveis, acessos à memória ou variações de latência.
2. Resposta reativa, para reagir dinamicamente a tentativas de evasão por meio de reinstrumentação seletiva ou redirecionamento de chamadas críticas.
3. Definição de métricas padronizadas de transparência e *overhead*, além da identificação falsos positivos com base em perfis de evasão detectados.

8. Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES-PROEX) - Código de Financiamento 001. Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas – FAPEAM – por meio do projeto POSGRAD 2024/2025.

Referências

Nethercote, N. (2004). Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory.

Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontana, L., Gritti, F., and Zanero, S. (2017). Measuring and defeating anti-instrumentation-equipped malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 73–96. Springer.

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

Santos Filho, A. and Feitosa, E. (2019). Reduzindo a superfície de ataque dos frameworks de instrumentação binária dinâmica. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, pages 253–266. SBC.