`VoiDbg`: Projeto e Implementação de um *Debugger* Transparente para Inspeção de Aplicações Protegidas

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

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

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

Abstract. *Debuggers are important tools for software development, as they support code inspection and, consequently, its validation. In systems security, debuggers can be used to assist malware analysis and reverse engineering, allowing researchers to investigate several execution paths. However, both legitimate software (for intellectual property protection) and malware (for detection avoidance) are often equipped with anti-debug techniques. Therefore, we need to accomplish transparency to overcome these techniques. To do so, we introduce* `VoiDbg`*, a novel debugger able to analyze protected software in a transparent way through hardware monitoring support. We present* `VoiDbg`*'s design and implementation, as well as tests to validate it.*

Resumo. *Debuggers são ferramentas importantes no desenvolvimento de software, pois auxiliam na inspeção de código e, com isso, sua validação. Em segurança de sistemas, debuggers podem ser usados em análise de malware e engenharia reversa, permitindo a investigação de vários caminhos de execução de aplicações. Entretanto, programas legítimos (para proteção de propriedade intelectual) e maliciosos (para evitar detecção) podem ser equipados com técnicas de anti-debug. Logo, sua inspeção deve ser feita transparentemente. Para tanto, introduz-se o* `VoiDbg`*, um debugger inovador capaz de analisar programas protegidos de modo transparente via suporte de monitoração em hardware. Além do projeto e implementação, apresenta-se testes de validação do* `VoiDbg`*.*

1. Introdução

Debuggers são ferramentas fundamentais para o processo de desenvolvimento e análise de *software*. Por meio de seu uso, pode-se identificar falhas de implementação no programa analisado, tais como *memory leaks*, *pointer dereference*, *use-after-free*, entre outras. Seu uso é também importante para as etapas de validação, na qual se pode verificar o correto funcionamento de partes do sistema, incluindo trechos críticos como tratamento de exceções e erros. Os *Debuggers* têm ainda um papel importante para segurança computacional, uma vez que podem ser aplicados a procedimentos de engenharia reversa de *software* em geral e análise de *malware*. No primeiro caso, pode-se inspecionar um dado programa em busca de "artefatos", como códigos indesejados, intrusivos e *backdoors*. No segundo, pode-se obter traços de execução do programa monitorado de forma a se compreender a abrangência de uma potencial infecção pelos caminhos possíveis de serem seguidos. Em ambos os casos, o uso do *debugger* auxilia a tomada das contra-medidas mais apropriadas.

 ©2016 SBC — Soc. Bras. de Computação

No entanto, o *software* moderno (e em especial o *malware*) conta com muitas técnicas de anti-análise, seja para efeitos de proteção de direitos autorais ou para evitar a detecção e alcançar seu objetivo malicioso. Tais técnicas, quando aplicadas contra um *debugger* tradicional, impedem ou dificultam a análise do programa. Assim, surgiram propostas na literatura de métodos e técnicas para monitoração transparente. Dessa forma, o *software* sob análise não deve perceber que a inspeção de seu funcionamento está ocorrendo, o que é uma extensão do princípio de Heisenberg da minimalidade da intrusão [Rosenberg 1996]. Dentre as diferentes formas de monitoração transparente, destacam-se aquelas que fazem uso de suporte de *hardware* para a extração de dados, dado que esta ocorre em um nível mais privilegiado do que o de execução do *software* sob análise. Neste artigo, introduz-se o `VoiDbg`, uma nova forma de implementar um *debugger* transparente contando com o auxílio do *hardware*. O protótipo proposto se utiliza de medidores de desempenho do processador para possibilitar a inspeção de aplicações executáveis no sistema operacional Windows 8.

O restante deste artigo está organizado como segue: a Seção 2 apresenta os requisitos do procedimento de *debugging* e uma breve revisão da implementação desses mecanismos, bem como os trabalhos relacionados; na Seção 3, introduz-se o funcionamento do mecanismo de *hardware* utilizado para controlar o processo de *debugging* transparente e discute-se as decisões de projeto e as características do `VoiDbg`; na Seção 4, discute-se as especifidades de implementação do `VoiDbg`; a Seção 5 mostra os testes e resultados para validação de transparência e funcionalidade do protótipo; o artigo é concluído na Seção 6.

2. *Debuggers*: visão geral e trabalhos relacionados

Esta seção apresenta os requisitos mínimos para o funcionamento de um *debugger* e analisa como as implementações atuais suprem os mesmos. Além disso, são mostrados trabalhos relacionados à criação de *debuggers* transparentes e não-convencionais.

2.1. Visão geral

De uma forma ampla, um *debugger* deve satisfazer os seguintes requisitos:

1. **Execução em pequenos passos (granularidade)**: O *debugger* deve permitir a execução/inspeção de código de forma granular, em passos tão pequenos quanto se queira—de *single-step* a chamadas de funções—a fim de compatibilizar a região de interesse ou alvo da depuração com as informações providas. Maior granularidade tem a vantagem de se poder inspecionar estados menores, identificando com maior precisão a causa da mudança de estados. Este requisito faz com que abordagens de coleta de dados baseadas em amostragem (*probing*) e simples suspensão de processos não sejam adequadas ao processo de *debugging*.
2. **Identificação de estados (ponto de parada)**: A execução deve garantir que o ponto de parada seja conhecido, isto é, que se tenha previsibilidade sobre a execução de código. O requisito de contexto é apresentado em [Rosenberg 1996]. Este, em conjunto com o requisito de granularidade, assegura que a região de interesse seja efetivamente inspecionada. Devido a este requisito, abordagens de amostragem e simples suspensão de processos não são adequadas ao processo de *debugging*, já que não asseguram a parada da execução na região de interesse.
3. **Inspeção de estados**: Dado um estado, deve-se ser capaz de obter informações de contexto sobre o mesmo. Isso possibilita a construção de um mapa da execução de código e transição de estados, que é a base da identificação de características

 ©2016 SBC — Soc. Bras. de Computação

do *software* sob análise. Tal procedimento pode ser realizado de diferentes maneiras, seja por instrospecção de máquina virtual, do sistema operacional ou mesmo via simples APIs de sistema, uma vez que o processo encontra-se com execução suspensa.

2.1.1. Implementações tradicionais

A seguir, implementações tradicionais de *debuggers* são revisitadas para que se possa contrastar as soluções vigentes com a solução aqui proposta. No âmbito deste artigo, as abordagens de *debug* são classificadas em: suporte do sistema operacional; injeção de código e emulação; e suporte de *hardware*.

Suporte do sistema operacional. Muitos sistemas operacionais fornecem facilidades e recursos para *debug*. Os sistemas GNU/Linux, por exemplo, possuem a interface `ptrace`[1] para controle de processos. Essa interface faz uso da arquitetura de processos do sistema (*fork*[2] + *copy on write*) para controlar os processos-filho , permitindo a execução passo-a-passo, a obtenção de dados de funções e *syscalls* e a instrumentação de código através de *callbacks* para eventos pré-determinados. Muitos utilitários do sistema são construídos sobre o `ptrace`, como o *debugger* GDB [GDB 2016] e o *tracer* `strace`[3]. No entanto, o `ptrace` não é transparente, podendo ser detectado com o uso de seu próprio *framework* (Listagem 1).

Listagem 1. Detecção de uso do ptrace.

```
1 if (ptrace(PTRACE_TRACEME, 0, NULL, 0) == -1)
2 printf("debugged!\n");
```

O sistema operacional Windows também possui facilidades para *debug*. Embora não possa fazer uso direto da arquitetura de processos—uma chamada de `CreateProcess` [Microsoft 2016a] não compartilha contexto com o processo filho—o sistema fornece uma série de APIs [Microsoft 2016c] para controlar outros processos. Um exemplo de uso destas APIs é o WinDbg [Microsoft 2016d], que também não é transparente, pois pode ser identificado pelo uso da API `IsDebuggerPresent` [Microsoft 2016i].

Injeção de código e emulação. Ferramentas como a Frida [Frida 2015] fazem uso da injeção de código para instrumentar *software*. A maior limitação desta abordagem é sua não-transparência devido à possibilidade de se detectar a alteração no fluxo através de um *hash* da própria memória do *software* sob análise. Ferramentas como o DynamoRIO [Bruening et al. 2012] e Valgrind [Nethercote and Seward 2003], por sua vez, fazem uso de emulação para a construção de um ambiente controlado no qual se pode inspecionar em detalhes aspectos como contexto e memória. Esta abordagem pode não ser transparente sob condições específicas.

Suporte de *hardware*. Além de suporte do sistema operacional e da instrumentação de código e emulação, pode-se contar com recursos do processador para a construção de um *debugger*. Processadores modernos [Intel 2011] contam com recursos como registradores de *debug*, *hardware breakpoints* e execução *single-step* via *trap flag*. *Debuggers* como Ollydbg [OllyDbg 2013] fazem uso deste recurso para inspecionar aplicações. Em-

[1] `http://man7.org/linux/man-pages/man2/ptrace.2.html`
[2] `http://linux.die.net/man/2/fork` [3] `http://linux.die.net/man/1/strace`

 ©2016 SBC — Soc. Bras. de Computação

bora mais eficiente, esta abordagem é limitada em transparência, uma vez que a *trap flag* altera *bytes* de instruções, dispara tratamento de exceções e define *bits* em registradores de controle.

2.2. Trabalhos relacionados

Devido a questões de transparência e do surgimento de novas linguagens e paradigmas de programação, esforços têm sido direcionados ao desenvolvimento de novos *debuggers*, cujos principais são mencionados a seguir.

Inspeção de código. Embora *debuggers* sejam soluções bem estabelecidas, seu nível de abstração ainda é concentrado nos estados do processador e valores de memória, tendo pouco suporte a construções de alto nível. Moldable [Chiş et al. 2015] tenta suprir as diferenças de abstração entre o baixo nível das informações e as construções de linguagens orientadas a objetos propondo um *framework* para lidar com informações de contexto. Outro problema recorrente em *debuggers* é lidar com sistemas multiprocessados/multi-*thread*. As soluções que atuam em paralelo sofrem de problemas de sincronização e de reconstrução de contexto, cuja abordagem data de 1996 [Hood 1996] e, desde então, tenta-se mitigá-los [Mäkelä et al. 2013, Schulz and Mueller 2000]. Várias das soluções propostas fazem uso de virtualização para monitorar o sistema via introspecção. PDB [Ho et al. 2004] usa o Xen para *debugging* em paralelo de *grids* computacionais, enquanto que [Ho and Hand 2005] propõem uma solução baseada em virtualização para lidar com construções de reuso de código. Um desafio recente é o uso de GPUs, pois seu modo de programação requer diferentes formas de *debug*. Para lidar com isto, [Sharif and Lee 2008] propõem um *framework* para interceptação de chamadas e reconstrução do fluxo até a exibição do pixel.

Análise de *malware*. As abordagens de novos *debuggers* para análise de *malware* focam, sobretudo, no aspecto da transparência. Dada a natureza evasiva dos programas maliciosos mais sofisticados, os *debuggers* para tal finalidade fazem uso de recursos de *hardware* para a inspeção. HyperDbg [Fattori et al. 2010] usa os recursos de virtualização do processador para implementar instrospecção de máquina virtual, *breakpoints* e inspeção de registradores e memória, que é capaz de realizar execução *single-step*. MALT [Zhang et al. 2015], por sua vez, usa o modo SMM do processador por meio da reescrita de BIOS para ter controle total do sistema. A maior limitação das abordagens citadas refere-se à sobrecarga de se ter que instrumentar todo um sistema e à dificuldade de implementação, dado que se requer a reescrita do BIOS ou de uma máquina virtual. A abordagem deste artigo busca ser uma versão aprimorada destas abordagens, isto é, com menor sobrecarga.

Abordagens não-tradicionais. Outros aspectos do processo de *debugging*, como a tarefa de *debug* em si, podem ser vistos em [Araki et al. 1991]. Para fins de proteção de código, técnicas de anti-debug têm emergido, como em [Yi et al. 2009], que faz uso de máquinas virtuais para a construção de um código equipado com técnicas de anti-análise. O experimento mostra a incapacidade do WinDBG inspecionar este código. As técnicas de anti-análise estão bem estabelecidas e disseminadas, com literatura específica para "contra-atacar" *debuggers* [Kaspersky 2007].

3. `VoiDbg`: Projeto

Para a construção de um *debugger* transparente, busca-se atender aos três requisitos básicos definidos na Seção 2.1. Contudo, evitamos utilizar as tecnologias apresentadas anteriormente, dadas as limitações expostas. A proposta deste artigo baseia-se em um recurso

 ©2016 SBC — Soc. Bras. de Computação

de *hardware*, a monitoração de *branches*, diferente dos apresentados anteriormente—*trap flag, debug register, virtual machine instruction set, SMM mode*.

Processadores modernos de diferentes fabricantes possuem variados medidores de desempenho, os quais possibilitam obter informações acerca do uso de memória, desempenho de cache, predição de *branch* e outros recursos. `VoiDbg` usa o mecanismo BTS (*Branch Trace Store*) dos processadores Intel em sua implementação. Tal mecanismo é um recurso do processador que armazena endereços origem e destino de desvios de fluxo de execução de código (instruções `JMP`, `CALL` e `RET`) em uma página de memória do sistema. O BTS permite ainda definir um *threshold* de armazenamento para o qual uma interrupção é gerada no processador, tornando-o um mecanismo adequado para realizar o controle de fluxo de execução de código no `VoiDbg`.

3.1. Decisões de projeto

Para atender o requisito de **execução em pequenos passos**, o mecanismo BTS foi aplicado de modo que a execução ocorra apenas em blocos entre duas instruções de desvio de fluxo. A implementação desse requisito deu-se pela definição de um *threshold* para o mecanismo BTS de uma única instrução de desvio. Esta abordagem tem a mesma ordem de grandeza de inspeção que o avanço em *single-step*.

Para atender o requisito de **identificação de estado**, o mecanismo BTS com interrupções também é considerado. A ocorrência destas garante que a execução esteja suspensa e, portanto, que o endereço fornecido pelo mecanismo seja o do bloco de execução corrente. Ademais, pode-se também reconstruir o contexto atual através da diferença entre o endereço provido na iteração anterior frente a atual.

Por fim, para a **inspeção do estado**, usam-se as APIs do próprio sistema operacional. A consistência dos dados é garantida pela suspensão da execução na interrupção.

A transparência de execução no sistema, entendida como a ausência de efeitos colaterais de execução, é garantida pelo fato da análise ser realizada em ambiente *bare-metal*, dada a necessidade de uso dos recursos específicos do processador, não acessíveis através de emulação por máquinas virtuais.

3.2. Modelo de ameaça, arquitetura da solução e fluxo de análise

A proposta deste artigo considera o *software* sob análise em execução no nível de espaço do usuário. Abordagens de *debug* em *kernel* envolvem um nível de instrumentação incompatível com a proposta de um *lightweight debugger*. Assume-se também que o objeto da análise atue em *single-core*, dado que a reconstrução do fluxo é realizada a partir dos dados do BTS em uma única CPU. A expansão para um cenário *multi-core* é deixada como trabalho futuro.

Além do programa monitorado, o analisador (*debugger*) também é executado em espaço de usuário, de modo a poder fazer uso das APIs fornecidas pelas bibliotecas de sistema. A coleta dos dados, contudo, só pode ser realizada em *kernel*, pois exige acesso a registradores especiais (MSRs). O tratamento da interrupção gerada pelo mecanismo BTS também só pode ser realizada em *kernel*. Deste modo, a coleta e o tratamento foram implementados através de um *driver* de *kernel*.

Para a comunicação de eventos e dados entre o *kernel* e o espaço de usuário são usadas rotinas de I/O. Porém, como a rotina de interrupção deve ser tratada instantaneamente para permitir a execução granular, o I/O deve ser implementada de forma assín-

 ©2016 SBC — Soc. Bras. de Computação

crona, gerando aviso ao cliente de *debug* no momento da ocorrência do evento. Tem-se então um fluxo em que o programa em execução realiza suas computações (Figura 1).

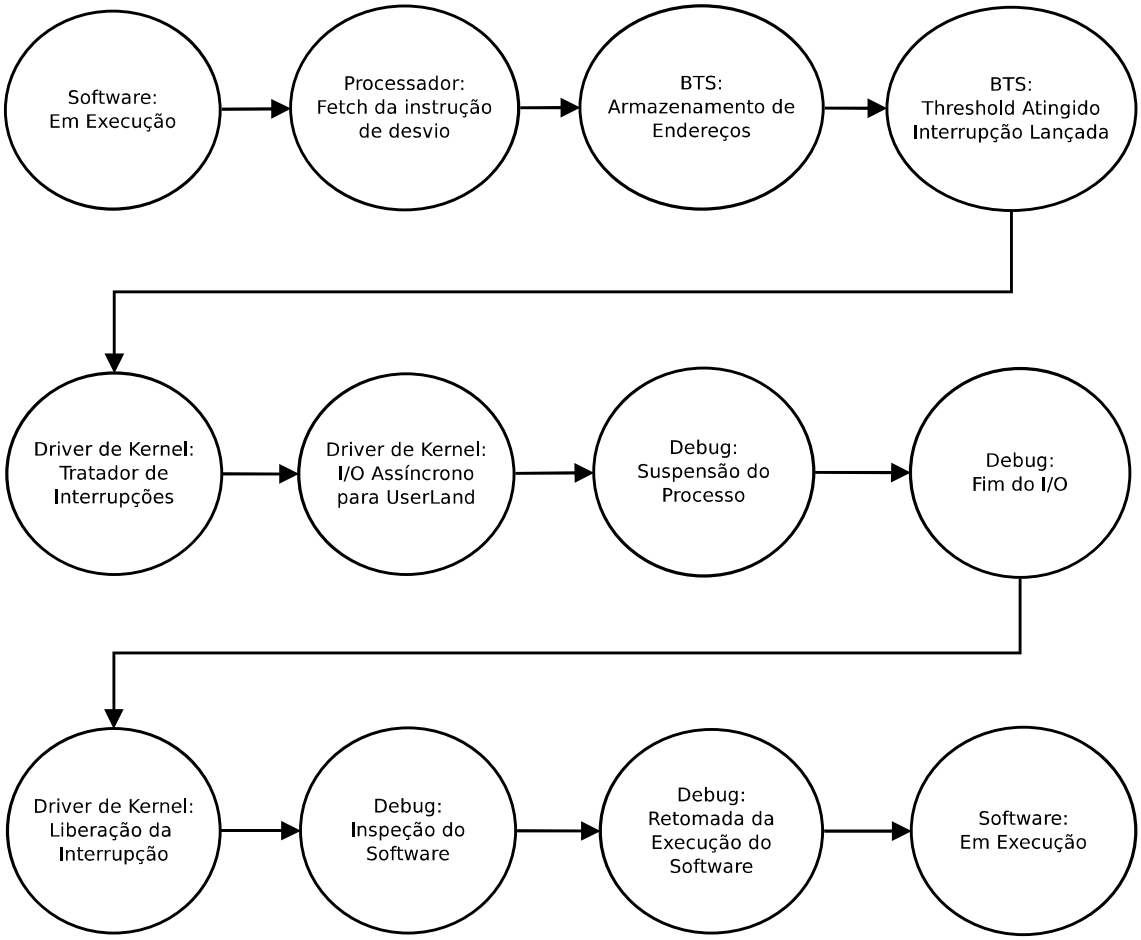

Figura 1. Fluxo de execução de um programa monitorado.

Em determinado momento da execução, o processador realiza o *fetch* de uma instrução de desvio relacionada às computações de um dado programa, cujos endereços de origem e destino são armazenados pelo mecanismo BTS. Uma vez que o *threshold* é de uma única instrução, um sinal de interrupção é disparado. O processo encontra-se em estado ativo, porém com sua execução suspensa devido à interrupção no processador. O sinal de interrupção é redirecionado pelo *kernel* para a rotina de tratamento do *driver*, a qual inicia uma chamada de I/O assíncrona para o cliente. O cliente recebe esta chamada e suspende a execução do processo em definitivo por meio de API do sistema. A chamada de I/O é finalizada, de modo que o *driver* possa terminar o tratamento de interrupção. Mesmo assim, o processo não é retomado pois seu estado mudou para "suspenso". Devido a esse estado, o cliente de *debug* pode realizar a inspeção do estado atual através das APIs do sistema. Após a obtenção de todos os dados desejados, resume-se a execução do processo, através de APIs do sistema, de modo a torná-lo ativo. O processo é retomado quando da geração da próxima interrupção.

3.3. Recursos de inspeção

O uso combinado do mecanismo BTS com as APIs de sistema permite às seguintes operações serem realizadas pelo `VoiDbg`:

 ©2016 SBC — Soc. Bras. de Computação

- Criação de um novo processo para ser inspecionado;
- Inspeção de um processo corrente;
- Suspensão de um processo;
- Retomada de um processo;
- Identificação do estado atual (funções/bibliotecas);
- Avanço ao próximo bloco;
- Leitura de valores de memória;
- Leitura das bibliotecas carregadas;
- Inspeção de registradores de contexto;
- Integração com outras soluções de *debuggers*.

4. `VoiDbg`: implementação

A seguir, descreve-se a implementação do `VoiDbg` em seus diversos níveis.

Captura de dados com BTS. O processo de configuração e a extração dos dados do mecanismo BTS são realizados diretamente em *kernel* por meio de um *driver*. A configuração é feita através de registradores especiais (*Model Specific Register* - MSR) e, para o armazenamento BTS, é fornecida uma página do sistema operacional.

Tratamento de interrupções. Utilizou-se no protótipo desenvolvido o tratamento de interrupção via *Non Maskable Interrupt* (NMI), cujo tratamento é obrigatório no instante em que o evento ocorre. Para isso, redireções na tabela *Advanced Programmable Interrupt Controler* (APIC) são necessárias. A configuração deve ser realizada em cada *core* caso o sistema seja multi-processado. O tratamento de uma NMI é regulado por um *watchdog* para que o processamento abusivo não cause congelamento do sistema. Desse modo, a única operação realizada dentro da NMI é o disparo da rotina de I/O assíncrono. Demais obtenções de dados são realizadas através de IOCTLs [Microsoft 2016e] após a suspensão do processo.

Disparo e suspensão de processos. O disparo de um novo processo a ser inspecionado é realizado diretamente pela chamada à função `CreateProcess` com o parâmetro CREATE_SUSPENDED. Já a suspensão de processos em execução, seja para anexação do *debugger* ou para sua inspeção após a ocorrência de uma interrupção, é feita através de APIs do sistema. Três métodos são conhecidos para a suspensão de processos: (i) a enumeração de todas as *threads* de um processo e a suspensão destas com a chamada à `SuspendThread` [Microsoft 2016n]; (ii) a chamada a uma função não documentada da biblioteca NTDLL (`NtSuspendProcess`); (iii) a chamada à `DebugActiveProcess` [Microsoft 2016b]. Dado que esta última pode ser identificada através da chamada à `IsDebuggerPresent` e que a primeira pode gerar um *deadlock* devido à não-sincronização da suspensão, a implementação do `VoiDbg` usa `NtSuspendProcess`.

Identificação do processo-alvo. O mecanismo BTS não fornece nenhum filtro no nível dos processos, fazendo com que todas as mudanças de fluxo de todos os processos em execução em uma dada CPU sejam armazenadas. Como o interesse deste trabalho reside nos dados de apenas um processo, faz-se necessário identificar quando o dado armazenado se refere a este. Para isso, o PID de interesse é carregado pelo cliente no *driver* via IOCTL e é comparado com o PID obtido da execução corrente através de `PsGetCurrentProcessId` [Microsoft 2016l].

Step-into vs. Step-over. Parte considerável da execução de um programa não se refere ao código presente em seu próprio binário, mas sim em bibliotecas dinamicamente

 ©2016 SBC — Soc. Bras. de Computação

ligadas. Estas executam em espaço de usuário, consequentemente ativando o mecanismo de captura BTS. Como as bibliotecas são mapeadas e relocadas dentro do espaço de endereçamento do processo, sua execução é entendida como execução do código monitorado. Assim, ao efetuar o *debugging* de um programa, inspeciona-se tanto o código do *software* quanto das bibliotecas utilizadas por ele em um modo chamado *step-into*. Para os casos em que se deseja inspecionar apenas o código do programa sem suas bibliotecas (chamado de modo *step-over*), deve-se adicionar filtros ao modo nativo de *debugging*.

I/O assíncrono. A comunicação da ocorrência do evento deve ser rápida, logo a escolha pela implementação de I/O assíncrono. No `VoiDbg`, usa-se a técnica conhecida como *Inverted call*, na qual o programa cliente solicita um dado (evento) que, se não estiver disponível no servidor (*driver*), faz com que a chamada de I/O seja colocada em uma fila. O programa cliente passa então a escutar em uma porta de I/O. O *driver*, por sua vez, quando obtém o dado requisitado (interrupção), retira o I/O pendente da fila e o dispara.

Identificação de estado (introspecção). O mecanismo BTS fornece a posição da memória de instruções na qual a execução do programa se encontra em um dado momento. Contudo, este resultado traz pouca informação semântica. Para aumentar o nível de informação provida, aplica-se um processo chamado de instrospecção, através do qual pode-se identificar a biblioteca ou função onde o programa está. Para a introspecção, inicialmente obtém-se o endereço de todas as bibliotecas carregadas no sistema por meio de enumeração [Microsoft 2016f] e *dump* de endereços [Microsoft 2016g] (necessário devido ao processo de aleatorização (ASLR)). Após identificar a biblioteca com endereço compatível com o atual, busca-se a função desta chamada pelo programa monitorado.

Acesso à memória. A inspeção de memória é um recurso que permite reconstruir o estado atual da execução. A obtenção de dados da região de memória do processo monitorado é realizada diretamente através da API `ReadProcessMemory` [Microsoft 2016m]. A criação do processo pelo `VoiDbg` assegura a obtenção de permissões para o procedimento citado.

Bibliotecas carregadas. A verificação de bibliotecas carregadas é uma etapa importante do processo de inspeção de *software*, pois pode indicar a função de um módulo inspecionado deste (pela ligação com uma DLL de criptografia, por exemplo) ou mesmo sua integridade (*hooks* de modo usuário são feitos por meio da injeção de uma biblioteca). A inspeção de módulos carregados é possível com o uso da API `GetModuleHandle`.

Valores de Contexto. Valores de contexto são importantes para a reconstrução/identificação do estado atual do processo. Tais valores incluem os dados nos registradores gerais, o apontador de pilha e as *flags* marcadas. A obtenção desses dados envolve a enumeração de todas as *threads* do processo monitorado e o uso da API `GetThreadContext` [Microsoft 2016h]. A consistência do estado é garantida, dada a suspensão da execução pelo *debugger*.

Integração com outras soluções. Embora o `VoiDbg` forneça um cliente remoto para a inspeção dos dados, a integração com soluções já consolidadas é importante para tornar possível o uso de ferramentas legadas e *scripts* feitos utilizando-as. Para tanto, foi implementada a integração de `VoiDbg` com o cliente do GDB, permitindo seu uso e de suas ferramentas. Pode-se, inclusive, efetuar a inspeção de um cliente Windows a partir de um sistema Linux. A implementação que serviu como referência para o *stub* do GDB compatível com o `VoiDbg` é a disponível em [mseaborn 2014], com a devida migração

 ©2016 SBC — Soc. Bras. de Computação

da plataforma Linux para Windows. A integração com GDB possibilita a inspeção de registradores, de memória e o avanço de *branches* através do comando `next`. O servidor que implementa o *stub* é responsável por converter o resultado provido pelas APIs do Windows para as estruturas de dados esperadas pelo GDB.

5. Testes e resultados

A seguir, são exibidos os resultados obtidos com os testes do `VoiDbg`, nos quais validou-se o funcionamento de todas as funcionalidades planejadas e avaliou-se sua transparência, desempenho e limitações. A Listagem 2 mostra a suspensão de um processo já em execução, enquanto que a Listagem 3 mostra a criação de um novo processo em estado suspenso. Já a Listagem 4 mostra o avanço da execução de *branch* em *branch*.

Listagem 2. Suspensão de processo.

```
1  4488 Suspended > _
```

Listagem 3. Criação de processo.

```
1 Sending Sample ToyProgram.exe
2 READ 7680 bytes
3 > Created suspended process PID 1268
```

Listagem 4. *Branch-by-branch steps*.

```
1 > n
2 > next branch
3 <0x1ba68fe9> BRANCH TO 0x33f2f267
4 > n
5 > next branch
6 <0x33f2f267> BRANCH TO 0x33f2e4b7
```

A Listagem 5 mostra a instrospecção do endereço destino. Neste, o endereço é identificado como sendo referente à biblioteca `MSVCR110D.dll`. A Listagem 6 mostra o *dump* de memória de 10 bytes a partir de um endereço interior ao processo sendo analisado.

Listagem 5. Introspecção.

```
1 > next branch
2 <0x1ba68fe9> BRANCH TO 0x33f2f267
3 > I
4 > Introspect To
5 > MSVCR110D.dll
```

Listagem 6. Valores de memória.

```
1 > dump memory
2 7f7d6d71005
3 10
4 > \xe9\x16\x00\x00\x00\xe9\x51\x00\x00\x00\xcc\xcc\xcc\xcc\xcc\
    xcc
```

A Listagem 7 mostra as bibliotecas carregadas pelo executável. Neste caso, apenas módulos do sistema são identificados. A Listagem 8 exibe a inspeção dos registradores de contexto.

 ©2016 SBC — Soc. Bras. de Computação

Listagem 7. Bibliotecas carregadas.

```
1  > Process 4488 suspended
2  > l
3  > list loaded libs
4  > C:\Windows\SYSTEM32\ntdll.dll
5  > C:\Windows\system32\KERNEL32.DLL
6  > C:\Windows\system32\KERNELBASE.dll
7  > C:\Windowz\SYSTEM32\MSVCR110D.dll
```

Listagem 8. Registradores de contexto.

```
1  > C
2  > Context
3  > R10 2
4  > R11 12851598
5  > R12 12851e00
6  > R13 0
7  > R14 0
8  > R15 12851e00
9  > RAX 1
10 > RBX 47e2f4e4
11 > RCX 0
12 > RDX 127f1bc
13 > RSP 47e2f3e8
14 > EFLAG 246
```

Para validar a integração do `VoiDbg` com o GDB, a porção de código ilustrada na Listagem 9 foi executada em uma sessão de *debugging* remoto. A Figura 2 mostra a exibição desses valores na sessão remota usando o GDB.

Listagem 9. Porção de código ASM.

```
1  int main()
2          __asm{mov eax,1}
3          __asm{mov ecx,2}
4          __asm{mov edx,3}
5          __asm{mov ebx,4}
```

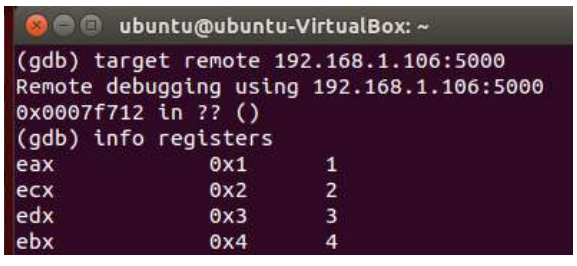

Figura 2. Integração com GDB.

5.1. Detecção do `VoiDbg`

Esta seção divide-se entre testes experimentais diretos e uma breve discussão sobre a transparência do protótipo implementado. Do ponto de vista teórico, o `VoiDbg` não é

 ©2016 SBC — Soc. Bras. de Computação

passível de injeções de código ou uso de APIs que possibilitem sua identificação. Para validar experimentalmente este ponto, foi criado um trecho de código que usa a API `IsDebuggerPresent`, exibido na Listagem 10. Este código foi submetido para execução no `VoiDbg` e o resultado é exibido na Figura 3, na qual pode-se observar que a anexação do *debugger* não foi detectada.

Listagem 10. Identificação de *debug*.

```
1       if(IsDebuggerPresent())
2               printf("debugged\n");
3       else
4               printf("NO DBG\n");
```

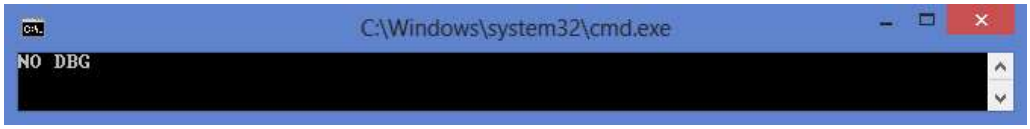

Figura 3. Detecção de *debugger* via API do SO.

Esta forma de detecção de *debuggers* é muito utilizada para a proteção de conteúdo proprietário, como em *game networks* [Woo and Kim 2012]. Um exemplo desta pode ser visto no executável da `Uplay`[4] que, quando submetido à execução em um *debugger*, produz o erro exibido na Figura 4. O uso de `VoiDbg` para análise do executável não apresentou o mesmo erro, permitindo o *debugging*, como exibido na Figura 5.

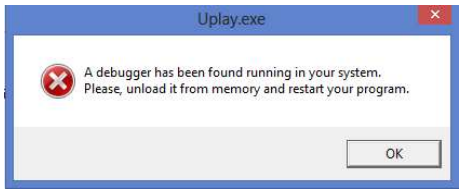

Figura 4. Detecção de *debugger* por uma aplicação legítima.

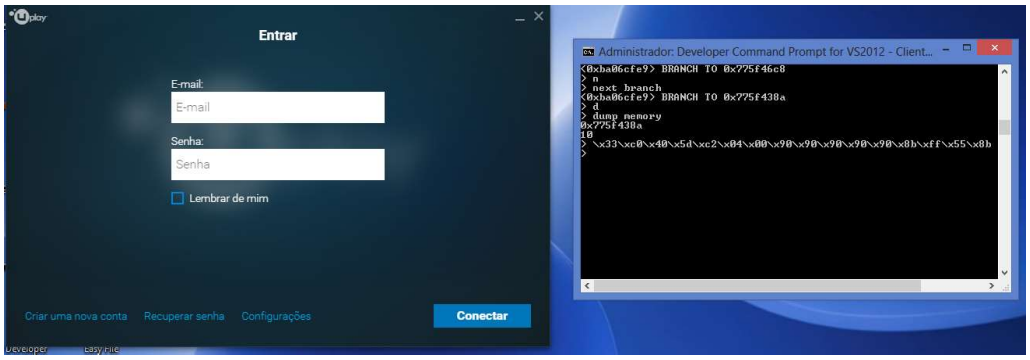

Figura 5. Uso do `VoiDbg` (dir.) com a aplicação legítima protegida (esq.).

Uma forma adicional de detecção poderia se dar através do registrador MSR, que ativa o mecanismo de monitoração de *branch*. Este registrador já é alvo de inspeção por *malware*, pois também é responsável por ativar o suporte de *debug* em *hardware*. Entretanto, os *bits* do mecanismo BTS e de *debug* são diferentes, permitindo a avaliação de que,

[4] `www.uplay.com`

 ©2016 SBC — Soc. Bras. de Computação

ao contrário dos *bits* de *debugger*, a ativação do mecanismo BTS por si só não pode ser entendida pelo *malware* como uma forma de monitoramento—diversas soluções utilizam-se deles, como o Intel VTune [Intel 2015], o Linux Perf [Linux 2015], e até monitores de desempenho da plataforma .Net [Microsoft 2016k] e do Windows [Microsoft 2016j].

Outra informação de *hardware* que poderia possibilitar a detecção de um processo de *debug* é a variação de tempo na execução. Dado que APIs de tempo do sistema são geralmente não-confiáveis, exemplares de *malware* podem fazer uso da medida de tempo do registrador de *timestamp counter* (TSC) para verificar a velocidade e continuidade de execução. Embora não se tenha abordado este tipo de contra-medida neste trabalho, esta já foi aplicada por outras soluções, como o HyperDbg e MALT, podendo assim ser abordada em um trabalho futuro.

5.2. Desenvolvimento e Portabilidade

O desenvolvimento da solução em sistema Windows exige apenas a escrita de um *driver* de *kernel*, aplicações em modo usuário a partir de bibliotecas pré-existentes e a utilização de um processador compatível com o monitor de *branch* utilizado, consistindo, de fato, em uma alternativa de menor custo de desenvolvimento em relação ao estado-da-arte (HVM e SMM).

Embora construído sobre o ambiente Windows, o sistema é portável para outras plataformas, já que o recurso de *hardware* é independente do sistema utilizado. Nestas plataformas, bibliotecas compatíveis devem ser utilizadas para os processos de introspecção e análise.

6. Conclusão

Neste artigo, apresentou-se o `VoiDbg`, um *debugger* leve e transparente com base no mecanismo BTS de monitoração de *branch* e em APIs do sistema. A solução proposta permite inspecionar o estado das aplicações em Windows 8 e possui integração com o cliente GDB para monitoração remota. O `VoiDbg` pode ser considerado leve em comparação com abordagens no estado-da-arte, pois não exige a reescrita do BIOS ou a construção de uma máquina virtual. O diferencial do trabalho consiste em ser uma solução cuja transparência é alcançada com o auxílio de recursos de *hardware*, permitindo até mesmo a inspeção de *software* com proteção *anti-debug*. Com isso, abre-se também a possibilidade de análise de *malware* equipado com mecanismos de anti-forense. Maiores informações sobre a solução proposta podem ser encontradas na páginas *web* da solução[5].

Agradecimentos

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

Referências

Araki, K., Furukawa, Z., and Cheng, J. (1991). A general framework for debugging. *IEEE Software*, 8(3):14–20.

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

 ©2016 SBC — Soc. Bras. de Computação

Bruening, D., Zhao, Q., and Amarasinghe, S. (2012). Transparent dynamic instrumentation. In *8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environments*, VEE '12, pages 133–144.

Chiş, A., Denker, M., Gîrba, T., and Nierstrasz, O. (2015). Practical Domain-specific Debuggers Using the Moldable Debugger Framework. *Comput. Lang. Syst. Struct.*, 44(PA):89–113.

Fattori, A., Paleari, R., Martignoni, L., and Monga, M. (2010). Dynamic and Transparent Analysis of Commodity Production Systems. In *Proc. IEEE/ACM Intl. Conf. on Automated Software Engineering*, ASE '10, pages 417–426, New York, NY, USA. ACM.

Frida (2015). Inject javascript to explore native apps. `http://www.frida.re/`.

GDB (2016). GDB: The GNU project debugger. `www.gnu.org/software/gdb`.

Ho, A. and Hand, S. (2005). On the Design of a Pervasive Debugger. In *Proc. Sixth Intl. Symp. on Automated Analysis-driven Debugging*, AADEBUG'05, pages 117–122, NY, USA. ACM.

Ho, A., Hand, S., and Harris, T. (2004). Pdb: pervasive debugging with xen. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 260–265.

Hood, R. (1996). The p2d2 project: Building a portable distributed debugger. In *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools*, SPDT '96, pages 127–136, NY, USA. ACM.

Intel (2011). Intel 64 and ia-32 architectures software developer's manual. `http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf`.

Intel (2015). Intel Vtune. `software.intel.com/en-us/intel-vtune-amplifier-xe`.

Kaspersky, K. (2007). *Hacker Disassembling Uncovered (Uncovered Series)*. A-List Publishing.

Linux (2015). Linux perf. `perf.wiki.kernel.org/index.php/Main_Page`.

Mäkelä, J.-M., Leppänen, V., and Forsell, M. (2013). Towards a parallel debugging framework for the massively multi-threaded, step-synchronous replica architecture. In *Proc. 14th Intl. Conf. Computer Systems and Technologies*, CompSysTech '13, pages 153–160, NY, USA. ACM.

Microsoft (2016a). Createprocess function. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425%28v=vs.85%29.aspx`.

Microsoft (2016b). Debugactiveprocess function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms679295%28v=vs.85%29.aspx`.

Microsoft (2016c). Debugging functions. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms679303%28v=vs.85%29.aspx`.

Microsoft (2016d). Debugging tools for windows. `https://msdn.microsoft.com/en-us/library/ff551063.aspx`.

Microsoft (2016e). Device input and output control (ioctl). `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa363219%28v=vs.85%29.aspx`.

Microsoft (2016f). Enumprocessmodules function. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682631%28v=vs.85%29.aspx`.

Microsoft (2016g). Getmodulehandle function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683199%28v=vs.85%29.aspx`.

Microsoft (2016h). Getthreadcontext function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms679362\%28v=vs.85\%29.aspx`.

 ©2016 SBC — Soc. Bras. de Computação

Microsoft (2016i). Isdebuggerpresent. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680345%28v=vs.85%29.aspx.

Microsoft (2016j). Performance counters. https://msdn.microsoft.com/pt-br/library/windows/desktop/aa373083\%28v=vs.85\%29.aspx.

Microsoft (2016k). Performancecounter class. msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter\%28v=vs.110\%29.aspx.

Microsoft (2016l). Psgetcurrentprocessid routine. https://msdn.microsoft.com/en-us/library/windows/hardware/ff559935%28v=vs.85%29.aspx.

Microsoft (2016m). Readprocessmemory function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680553%28v=vs.85%29.aspx.

Microsoft (2016n). Suspendthread function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms686345%28v=vs.85%29.aspx.

mseaborn (2014). gdb-debug-stub. github.com/mseaborn/gdb-debug-stub.

Nethercote, N. and Seward, J. (2003). Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44 – 66.

OllyDbg (2013). Ollydbg. www.ollydbg.de.

Rosenberg, J. B. (1996). *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA.

Schulz, D. and Mueller, F. (2000). A thread-aware debugger with an open interface. In *Proc. 2000 ACM SIGSOFT Intl. Symp. Software Testing and Analysis*, ISSTA '00, pages 201–211.

Sharif, A. and Lee, H.-H. S. (2008). Total recall: A debugging framework for gpus. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 13–20, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Woo, J. and Kim, H. K. (2012). Survey and research direction on online game security. In *Proceedings of the Workshop at SIGGRAPH Asia*, WASA '12, pages 19–25.

Yi, T., Zong, A., Yu, M., Gao, S., Lin, Q., Yu, P., Ren, Z., and Qi, Z. (2009). Anti-debugging framework based on hardware virtualization technology. In *Research Challenges in Computer Science, 2009. ICRCCS '09. International Conference on*, pages 218–220.

Zhang, F., Leach, K., Stavrou, A., Wang, H., and Sun, K. (2015). Using hardware features for increased debugging transparency. In *IEEE Symp. Security and Privacy (SP)*, pages 55–69.

 ©2016 SBC — Soc. Bras. de Computação