

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

Marcus Botacin<sup>1</sup>, André Grégio<sup>1,2</sup>, Paulo Lício de Geus<sup>1</sup>

<sup>1</sup>Instituto de Computação - UNICAMP  
{marcus,paulo}@lasca.ic.unicamp.br

<sup>2</sup>Universidade Federal do Paraná (UFPR)  
gregio@inf.ufpr.br

09 de Novembro de 2016

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Panorama

## *Debuggers*

- Inspeção.
- Validação.
- Descoberta de Falhas.

## Cenário Atual

- Injeção de código.
- Efeitos Colaterais.
- Anti-análise.

## Objetivos

- Desenvolvimento de solução transparente.

# Tópicos

- 1 Parte I
  - Introdução
  - **Requisitos**
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Requisitos

## Execução em pequenos passos

- Granularidade
- Exige mais que *probing*.

## Identificação de estados

- Previsibilidade.

## Inspeção de estados

- Obtenção de contexto.

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - **Implementações**
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Implementações Atuais

## Suporte de sistema operacional

- APIs de sistema (*ptrace*).
- Detectável pela própria API.

## Injeção de código/emulação

- *Frida*, *DynamoRIO*, *Valgrind*.
- Efeitos colaterais.

## Suporte de *Hardware*

- *OllyDbg*.
- *Hardware-breakpoints*.
- Detecção de registradores.



# Evasão de análise

## Evasão de ptrace

Listagem 1 : Detecção de uso do ptrace.

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

## Evasão no ambiente Windows

Listagem 2 : Identificação de debug.

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

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - **Trabalhos Relacionados**
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Trabalhos Relacionados

## Inspeção de código

- Construções de alto-nível (POO).
- Sistemas distribuídos, *grids*, GPUs.

## Análise de *malware*

- *Hardware Virtual Machines* (HVM).
- *System Management Mode* (SMM).
- Análise Transparente (Quantos ?).

## Abordagens não-tradicionais

- Anti-Debug.

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Monitoração de *branch*

## Definição

- Recurso do processador
- Armazenamento em registradores ou em memória.

## Funcionamento

- Interrupção ao atingir *threshold*.
- Filtragem de ações e por privilégio.
- Acesso via *kernel*.

## *Branch*

- JNE, JMP, CALL, RET.
- Tratamento de Exceções.

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - **Proposta**
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Proposta de solução.

## Funcionamento

- *Threshold* de 1 desvio.
- Interrupção para isolamento de processos.
- Introspecção de sistema para obtenção de contexto.

## Atendendo os Requisitos

- Granularidade: *Branch a Branch*.
- Identificação de Estado: Interrupção.
- Contexto: Introspecção.

## Modelo de Ameaças

- Nível de usuário.
- *Single-Core*.

## Fluxo de Análise.

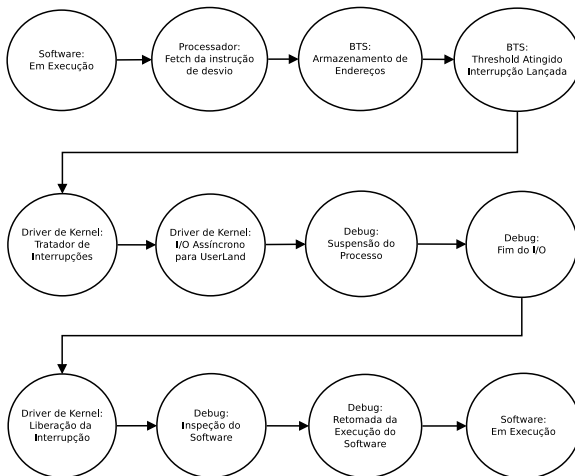


Figura : Fluxo de execução de um programa monitorado.



## Recursos de Inspeção

- 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*.

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - **Implementação**
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Implementação

## Captura de Dados

- *Model Specific Register (MSR) exige driver de kernel.*

## Tratamento de Interrupções

- *Non Maskable Interrupt (NMI).*

## Suspensão de processos

- SuspendThread.
- DebugActiveProcess.
- SuspendProcess.

# Implementação

## Identificação de Processos

- `PsGetCurrentProcessId`

## *Step-Into vs. Step-Over*

- Filtragem de ações intermediárias.

## Acesso à memória

- `ReadProcessMemory`.

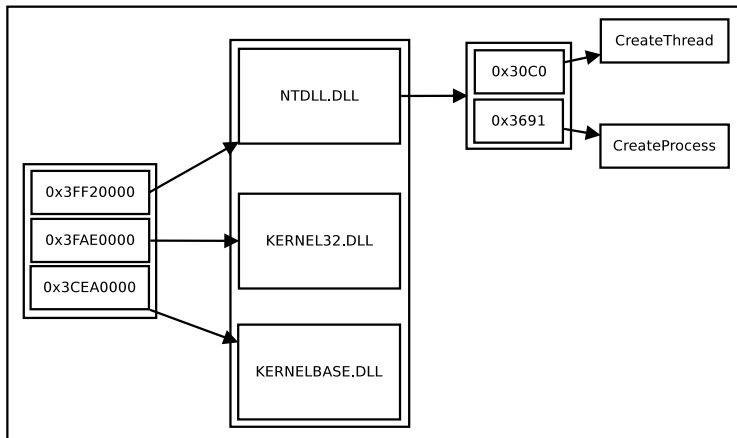
## Bibliotecas Carregadas

- `GetModuleHandle`

## Contexto

- `GetThreadContext`

# Introspecção.

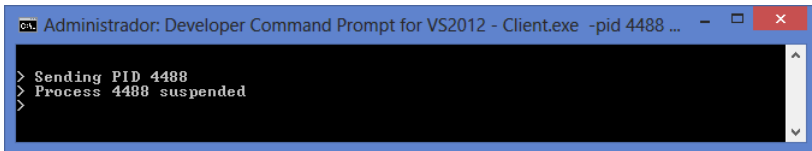


**Figura :** Diagrama de funcionamento do mecanismo de introspecção para associação de nomes de funções a endereços de módulos.

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - **Resultados**
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

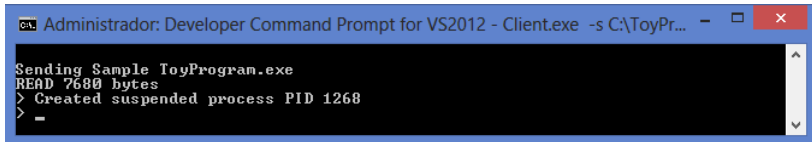
## Testes.



Administrador: Developer Command Prompt for VS2012 - Client.exe -pid 4488 ...

```
> Sending PID 4488
> Process 4488 suspended
>
```

Figura : Suspensão de processo.

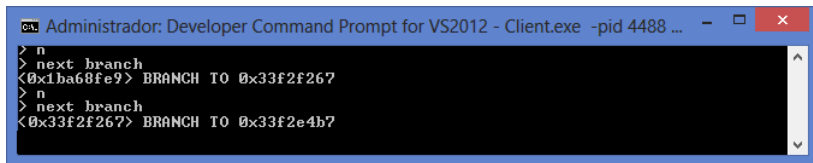


Administrador: Developer Command Prompt for VS2012 - Client.exe -s C:\ToyPr...

```
Sending Sample ToyProgram.exe
READ 7680 bytes
> Created suspended process PID 1268
> _
```

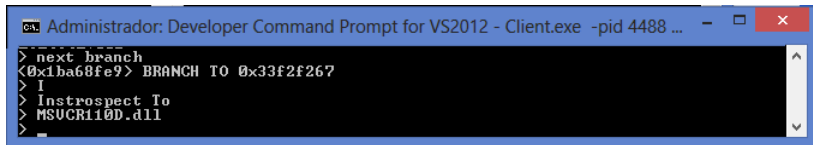
Figura : Criação de Processo.

# Testes.



```
Ca. Administrador: Developer Command Prompt for VS2012 - Client.exe -pid 4488 ...  
> n  
> next branch  
<0x1ba68fe9> BRANCH TO 0x33f2f267  
> n  
> next branch  
<0x33f2f267> BRANCH TO 0x33f2e4b7
```

Figura : *Branch-by-branch steps.*

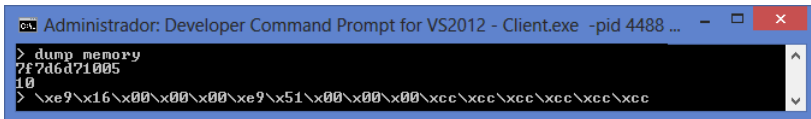


```
Ca. Administrador: Developer Command Prompt for VS2012 - Client.exe -pid 4488 ...  
> next branch  
<0x1ba68fe9> BRANCH TO 0x33f2f267  
> I  
> Instrospect To  
> MSUCR110D.dll  
> _
```

Figura : Introspecção no endereço alvo.



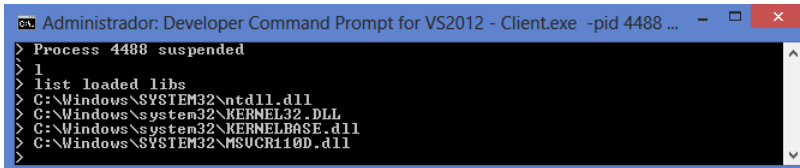
## Testes.



Administrador: Developer Command Prompt for VS2012 - Client.exe -pid 4488 ...

```
> dump memory
7f7d6d71005
10
> \xe9\x16\x00\x00\x00\xe9\x51\x00\x00\x00\xcc\xcc\xcc\xcc\xcc
```

Figura : Leitura de Memória.



Administrador: Developer Command Prompt for VS2012 - Client.exe -pid 4488 ...

```
> Process 4488 suspended
> 1
> list loaded libs
> C:\Windows\SYSTEM32\ntdll.dll
> C:\Windows\system32\KERNEL32.DLL
> C:\Windows\system32\KERNELBASE.dll
> C:\Windows\SYSTEM32\MSUCR110D.dll
```

Figura : Bibliotecas Carregadas.

# Testes.

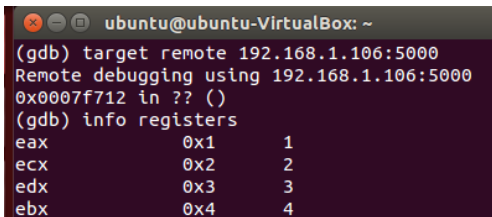
```
Administrador: Developer Command Prompt for VS2012 - Client.exe -a 127.0.0....
> C
> Context
> R10 2
> R11 12851598
> R12 12851e00
> R13 0
> R14 0
> R15 12851e00
> RAX 1
> RBX 47e2f4a4
> RCX 0
> RDX 127f71bc
> RSP 47e2f3e8
> EFLAG 246
>
```

Figura : Registradores de contexto.

## Integração com GDB.

### Listagem 3 : 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}
```



The screenshot shows a terminal window titled 'ubuntu@ubuntu-VirtualBox: ~'. The GDB session includes the following commands and output:

```
(gdb) target remote 192.168.1.106:5000  
Remote debugging using 192.168.1.106:5000  
0x0007f712 in ?? ()  
(gdb) info registers  
eax             0x1         1  
ecx             0x2         2  
edx             0x3         3  
ebx             0x4         4
```

Figura : Integração com GDB.

## Detecção por Aplicações Reais.

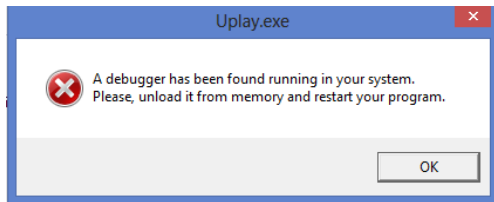


Figura : Detecção de *debugger* por uma aplicação legítima.

## Detecção por Aplicações Reais.

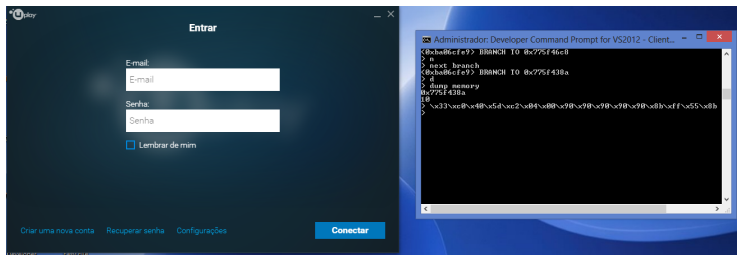


Figura : Uso do VoIDbg (dir.) com a aplicação legítima protegida (esq.).

# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - **Considerações Finais**
  - Conclusões e Agradecimentos

# Análise dos Resultados

## Sumarização

- Aplicações Protegidas.
- Soluções precisam ser transparentes.
- Mecanismo de monitoração de *Branch* pode ser utilizado.
- Análise é transparente.
- Validação com exemplares benignos reais.

# Limitações e Trabalhos Futuros

## Limitações

- Análise em *kernel*.
- Uso de APIs do sistema.

## Trabalhos Futuros

- Expansão das informações de contexto.
- Ampliação da integração com ferramentas.
- Aplicação a exemplares de *malware* evasivos.



# Tópicos

- 1 Parte I
  - Introdução
  - Requisitos
  - Implementações
  - Trabalhos Relacionados
- 2 Parte II
  - Fundamentos
  - Proposta
  - Implementação
  - Resultados
- 3 Parte III
  - Considerações Finais
  - Conclusões e Agradecimentos

# Conclusões

## Conclusões

- Mecanismos de análise transparente precisam ser desenvolvidos.
- Uso dos monitores de performance é viável.
- Modos de implementação de recursos de inspeção precisam ser estudados.

# Agradecimentos

- CNPq, pelo financiamento via Proj. MCTI/CNPq/Universal-A edital 14/2014 (Processo 444487/2014-0)
- CAPES, pelo financiamento via Proj. FORTE - Forense Digital Tempestiva e Eficiente (Processo: 23038.007604/2014-69).
- Instituto de Computação/Unicamp
- Departamento de Informática/UFPR

## Contato:

marcus@lasca.ic.unicamp.br  
paulo@lasca.ic.unicamp.br  
gregio@inf.ufpr.br

