UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MÓDULO DE CAPTURA, FILTRAGEM E REDIRECIONAMENTO DE MENSAGENS DIRECTX

GIOVANI DORVAL CIQUELERO CHAVES

BLUMENAU
2009

2009/2-09

GIOVANI DORVAL CIQUELERO CHAVES

MÓDULO DE CAPTURA, FILTRAGEM E REDIRECIONAMENTO DE MENSAGENS DIRECTX

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos, Dr - Orientador

BLUMENAU
2009

2009/2-09

MÓDULO DE CAPTURA, FILTRAGEM E REDIRECIONAMENTO DE MENSAGENS DIRECTX

Por

GIOVANI DORVAL CIQUELERO CHAVES

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. Mauro Marcelo Mattos, Dr. – Orientador, FURB
	_____
Membro:	Prof. Adilson Vahldick, Mestre – FURB
	_____
Membro:	Prof. Antonio Carlos Tavares, Mestre – FURB

Blumenau, 15 de dezembro de 2009

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, por ter criado o homem. Ao homem, por ter criado a máquina. À máquina, por prover o meu sustento.

Ao colega Victor Arndt Mueller, que foi de ajuda essencial na concepção da minha ideia.

Ao colega Amando Junior, que forneceu seu *precious* Logitech G25 ajudando a tornar a apresentação ainda mais interativa.

Ao professor Miguel Wisintainer pela ajuda com o acoplamento do hardware do BodyStruck à USB.

Ao meu orientador, Mauro Marcelo Mattos, por ter acreditado na conclusão deste trabalho (quando nem mesmo eu acreditava que daria conta).

Ao meu chefe, Josias Rafael Wagner, por sua infindável paciência com minhas incontáveis horas de expediente dedicadas à conclusão deste trabalho.

“Todos somos ignorantes, apenas em assuntos diferentes.”

Will Rogers

RESUMO

O presente trabalho descreve um sistema de captura, filtragem e redirecionamento de mensagens DirectX em ambiente Windows, utilizando a técnica de DLL *injection* através de *Hooks* para injetar código e modificar o fluxo normal de processamento dos eventos de Force Feedback no módulo DirectInput. A técnica de DLL *injection*, além do uso proposto neste trabalho (estender funcionalidades de um aplicativo fechado), também é um dos principais métodos de desenvolvimento de *malwares* e deve ser utilizada com responsabilidade. O projeto foi implementado e testado em um ambiente de jogo onde as ações de Force Feedback foram redirecionadas para dois *joysticks* USB instalados no mesmo computador.

Palavras-chave: *Force Feedback*. DirectX. DLL *injection*.

ABSTRACT

This work describes a system to capture, filtering and redirecting DirectX messages in Windows environment, using DLL injection technique through Hooks to inject code and modify the normal processing path of Force Feedback events on DirectInput module. The DLL injection technique, besides the proposed use in this work (extending closed application features), is one of the main methods to develop malwares and have to be used with responsibility. The project was implemented and tested using a game where Force Feedback actions were redirected to two joysticks installed in the same computer.

Keywords: *Force Feedback*. DirectX. DLL *injection*.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

CLR – *Common Language Runtime*

COM – *Component Object Model*

DCIE – *Detecting the Code Injection Engine*

DLL – *Dynamic Link Library*

GUID – *Globally Unique IDentifier*

HID – *Human Interface Device*

IAT – *Import Address Table*

OSF – *Open Software Foundation*

SDK – *Software Development Kit*

SO – Sistema Operacional

UML – *Unified Modeling Language*

UUID – *Universal Unique IDentifier*

LISTA DE SÍMBOLOS

% - por cento

- sustenido

SUMÁRIO

1 INTRODUÇÃO

Entre as tecnologias recentes que são aplicadas ao desenvolvimento de *games*[1], a realidade virtual vem ganhando atenção pela sua eficácia em transmitir para o mundo real as ações que ocorrem dentro dos jogos (CHOI; CHANG; KIM, 2004, p. 325). O principal objetivo do desenvolvimento de gráficos cada vez mais realistas em jogos é proporcionar ao jogador uma maior imersão, dando a impressão de que ele realmente está vivendo a situação apresentada pela problemática do *game*. Não é de se estranhar que jogadores tenham as mais diversas reações enquanto jogam: alguns inclinam-se durante curvas em jogos de corrida, outros levam sustos em *games* de suspense/ação ou ainda, tentam se esquivar de balas nos *shooters*[2].

Porém, não apenas gráficos extremamente detalhados e sons de ambientação são recursos utilizados para proporcionar a sensação de se estar dentro do *game*. Existem cabines e simuladores que abrigam o jogador para que este possa se sentir de fato pilotando um avião, nave ou carro. Ao se movimentar, estes equipamentos interagem com outros sentidos do jogador (equilíbrio, tato), resultando em uma resposta mais rápida no reconhecimento do cenário à sua volta (CHOI; CHANG; KIM, 2004, p. 325). Entretanto, *hardwares* simuladores muitas vezes são grandes e caros suficientemente para que sejam utilizados apenas para fins profissionais, que exigem obrigatoriamente maior aproximação com eventos reais (como no caso de treinamento de pilotos comerciais ou militares).

Todavia, há acessórios mais simples e portáteis (como *joysticks*[3] e volantes que reagem a determinadas ações do jogo oferecendo resistência ao movimento - o chamado *Force Feedback*[4], óculos estereoscópicos para simular 3D, pistolas e revólveres para jogos de tiro, entre outros), que são amplamente utilizados por jogadores (MARSHALL; WARD; MCLOONE, 2006). Contudo, os acessórios mais populares comercializados para *gamers* e que exploram o sentido do tato são *joysticks* e volantes que possuem a tecnologia *Force Feedback* implementada. A interação com estes equipamentos é feita através das mãos, consequentemente estimulando apenas esta parte do corpo, deixando de lado todas as outras

---

[1] *Game*: jogo eletrônico.

[2] *Shooters*: jogos de tiro em primeira pessoa.

[3] *Joysticks*: acessórios controladores de movimentos e ações para *games*.

[4] *Force Feedback*: tecnologia proprietária da Microsoft, com propósito de interagir com o usuário através de acessórios que vibram ou oferecem resistência ao movimento.

áreas que seriam estimuladas em situações reais.

Segundo Langweg (2004), o Microsoft DirectX é um grupo de tecnologias projetadas pela Microsoft para tornar computadores baseados no sistema operacional Microsoft Windows uma plataforma ideal para execução de aplicações com alta interação com o usuário, como jogos e apresentações multimídia. O DirectX é parte constituinte dos sistemas Windows 98, Windows Me e Windows 2000/XP. O DirectX possibilita aos desenvolvedores um conjunto consistente de *Application Programming Interfaces* (APIs) que disponibilizam acesso personalizado aos mais variados modelos e tipos de *hardware*. Estas APIs controlam o que é chamado de funções de baixo nível incluindo: gerenciamento de memória, renderização de gráficos e suporte para dispositivos de entrada como *joysticks*, teclado e mouse. Estas funções de baixo nível são agrupadas nos seguintes componentes principais: Microsoft Direct3D, Microsoft DirectDraw e Microsoft DirectInput.

Conforme Palmenäs (2008), em geral há duas formas para injetar código em um processo: *Dynamic Link Library* (DLL) *injection* ou *memory injection*. Na técnica de DLL *injection* força-se a aplicação a carregar do disco e executar uma DLL específica, enquanto na técnica de *memory injection* aloca-se uma área de memória no processo alvo e remotamente escrevem-se dados e código nesta área quando necessário. Após o processo de injeção uma nova *thread* é iniciada para executar o código injetado.

O presente trabalho tem como objetivo usar a técnica de DLL *injection* para desenvolver um módulo de captura e filtragem de mensagens DirectX geradas durante a execução de *games* que se comunicam com dispositivos *Human Interface Devices* (HIDs) como mouses e *joysticks*; e efetuar o redirecionamento destas mensagens. Desta forma, abre-se caminho e viabiliza-se a construção de novos tipos de acessórios que auxiliam na imersão do jogador reagindo aos estímulos desencadeados pela sua interação com o jogo através da tecnologia *Force Feedback*.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um módulo de captura, filtragem e redirecionamento de mensagens DirectX.

Os objetivos específicos do trabalho são:

a) disponibilizar um módulo de interceptação de mensagens do DirectX através de

DLL *injection* e filtragem dos eventos que envolvem *Force Feedback* no módulo `DirectInput`;

b) disponibilizar um aplicativo que permita a instalação e remoção do *Hook*[5] que fará a injeção da DLL customizada no processo alvo.

1.2 ESTRUTURA DO TRABALHO

O primeiro capítulo apresenta uma introdução do trabalho, os objetivos a serem alcançados e a estrutura do trabalho.

O segundo capítulo contempla a fundamentação teórica do trabalho e descreve o *Component Object Model* (COM), seguido pelo detalhamento do DirectX com enfoque no módulo `DirectInput`, o conceito de DLL *injection* e *Hooks*. Além disto, descreve a APIHijack, utilizada para o desenvolvimento da ferramenta apresentada e também mostra trabalhos correlatos.

O terceiro capítulo expõe a implementação dos principais pontos que fazem a filtragem e redirecionamento das mensagens, demonstrando e justificando os comandos utilizados.

Para finalizar, o quarto capítulo descreve a que conclusões este trabalho levou, e sugere extensões para o mesmo.

[5] *Hook*: mecanismo de interceptação de mensagens da fila do sistema operacional.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo explica cada uma das principais tecnologias e metodologias envolvidas no processo de desenvolvimento da ferramenta proposta, como COM, DirectX e DLL *injection*. Explica também o que são *Hooks* (utilizados no mecanismo de DLL *injection*) e a API utilizada para abstrair o remapeamento dos métodos do DirectX para os da DLL injetada.

2.1 COMPONENT OBJECT MODEL (COM)

Segundo Matos (2004, p. 8), o COM é um sistema que permite desenvolver objetos (componentes binários) que podem interagir com outros objetos. Os objetos COM podem ser utilizados e desenvolvidos em diversas linguagens de programação, sendo o Microsoft Visual C++ uma das preferidas por oferecer mecanismos que simplificam a implementação dos mesmos. Esta tecnologia oferece suporte ideal para o desenvolvimento de aplicações robustas, portáveis e escaláveis, por ser independente de plataforma e local, podendo os objetos rodarem em uma máquina remota pela rede.

A tecnologia COM é baseada na tecnologia de *Dynamic Link Library* (DLL), a qual constitui-se numa forma de armazenar funções pré-compiladas para posterior ligação em tempo de execução das aplicações (HUNT; BRUBACHER, 1999, p. 2). O Quadro 1 apresenta uma comparação entre os conceitos usados em programação tradicional e na tecnologia COM.

Conceito	Convencional (C++/OO)	COM
Cliente	Um programa que requisita um serviço de um servidor.	Um programa que chama métodos COM.
Servidor	Um programa que fornece serviços a outros programas.	Um programa que torna os objectos COM disponíveis a clientes COM.
Interface	Nenhum	Um apontador para um grupo de funções que são chamadas através do COM.
Classe	Um tipo de dados. Define um grupo de métodos e dados que são utilizados em conjunto.	A definição de um objecto que implementa uma ou mais interfaces COM. Também conhecido como coclass.
Objecto	Uma instância de uma classe.	Uma instância de uma coclass,
Marshalling	Nenhum	Transferência de dados entre o cliente e o servidor.

Fonte: Matos (2004).

Quadro 1 – Comparação entre a terminologia convencional e a usada no COM

Os objetos COM podem disponibilizar um conjunto de dados e um conjunto de funções relacionadas, de acesso e manipulação desses dados; a isto denomina-se Interfaces. Para se ter acesso aos métodos de uma Interface, é necessária a obtenção de um ponteiro para a referida interface (MATOS, 2004, p. 13). Para manter compatibilidade com versões anteriores, quando uma nova versão de um objeto COM é liberada, as antigas interfaces são preservadas e as novas funcionalidades liberadas através de uma nova interface (RABER; LASPE, 2007).

O Quadro 2 descreve a sequência de passos a serem seguidos para a criação de um objeto COM.

1	Criar um objecto COM e dar-lhe um nome. Este objecto será implementado dentro de um servidor COM.
2	Definir a interface e dar-lhe um nome.
3	Definir a função na interface e dar-lhe um nome.
4	Instalar o servidor COM.

Fonte: Matos (2004).

Quadro 2 – Passos para criar um objeto COM

Cada um dos servidores e objetos foram desenvolvidos por programadores diferentes, que podem aleatoriamente escolher nomes iguais para estes componentes. Como assegurar que tenham nomes diferentes e evitar colisões?

Cada objeto COM possui um GUID. Este identificador é montado através de um algoritmo que combina o endereço da rede, a data (com intervalos de 10 nanosegundos) e um contador. O resultado é um número de 128bits único que a Open Software Foundation (OSF), desenvolvedora do algoritmo, chama de *Universal Unique IDentifier* (UUID). A Microsoft utiliza o mesmo algoritmo para nomear os componentes do COM, apenas rebatizando para *Globally Unique IDentifier* (MATOS, 2005, p. 17). A convenção para escrita de GUIDs é em hexadecimal (Quadro 3). “Como não existe um tipo de dados em C++ de 128-bits, é usada uma estrutura para a sua representação. Essa estrutura consiste em quatro campos diferentes. Normalmente, nunca será necessário manipular cada um dos campos individualmente” (MATOS, 2004, p. 17), como pode ser visto no Quadro 4.

{929DBD21-AE6C-11d5-9239-00001CD1D262}

Fonte: Matos (2004).

Quadro 3 – Formato do GUID

```
Typedef struct _GUID
{
      unsigned long Data1;       // 4 bytes +
      unsigned short Data2;      // 2 bytes +
      unsigned short Data3;      // 2 bytes +
      unsigned char Data4[8];    // 8 bytes = 16 bytes = 128 bits
} GUID;
```

Fonte: Matos (2004).

Quadro 4 – Estrutura do GUID

Os objetos COM são gerenciados por servidores COM. Cada máquina no universo COM conecta-se com várias outras, formando uma estrutura em rede. Desse modo, é possível disponibilizar os serviços de uma máquina para toda a rede.

2.2 DIRECTX

Segundo Microsoft (2009a), "o DirectX é uma API multimídia que oferece uma interface padrão para interagir com elementos gráficos, placas de som e dispositivos de entrada, entre outros". Com seu surgimento em 1995, ele serve como um intermediário entre o *software* e o *hardware*, para que programadores possam abstrair as interações específicas com os diferentes conjuntos de *hardware* instalados na máquina (placa de som, *mouse*, teclado, etc).

> Quando os programadores escrevem *software* compatíveis com o DirectX, este *software* certamente funcionará em um *hardware* que também seja compatível com o DirectX. O DirectX torna mais fácil a vida dos desenvolvedores, fornecendo APIs. (STAROSCIK, 2003).

Originalmente, a API foi desenvolvida somente para as linguagens C e C++, sendo posteriormente lançada uma versão gerenciada que dava abertura para linguagens compatíveis com *Common Language Runtime* (CLR), C# ou VB.NET. Isso acabou gerando discussões sobre a performance da biblioteca gerenciada nos jogos. Porém, é possível usar uma combinação de código gerenciado e não-gerenciado, em jogos com necessidades de muito poder de processamento. De acordo com Microsoft (2009a), "escrever em código gerenciado torna os desenvolvedores mais produtivos, permitindo que eles criem código em maior quantidade e com mais segurança".

Desta forma, é utilizado um conjunto de instruções comuns que serão traduzidas por ele em comandos específicos de *hardware*, sendo constituído por diversos módulos com áreas bem definidas (Quadro 5).

Módulo	Descrição
DirectDraw	É uma biblioteca de funções gráficas em 2D.
Direct3D	É um conjunto de APIs para gráficos em 3D. Torna fácil para programadores inserirem efeitos sofisticados que eram inimagináveis alguns anos atrás. Quando uma placa em 3D é projetada para suportar uma versão do DirectX, o fabricante, na maioria das vezes, recorre aos recursos do Direct3D, tais como o mapeamento de deslocamento e as variações de pixel.
DirectSound	Suporta entrada e saída de som, incluindo o DirectSound3D para efeitos espaciais avançados, como o posicionamento de som em 3D e ecos em tempo real.
DirectMusic	É uma rica biblioteca de funções sonoras especificamente para música, permitindo que jogos criem sinfonias interativas e mesclem diversos temas musicais. Também é útil para efeitos sonoros ambientais.
DirectInput	Permite que programadores utilizem desde *joysticks* até mouses *Force Feedback*. Ele fornece até mesmo uma interface de calibragem para o controlador.
DirectPlay	É uma série de ferramentas de rede para jogos de múltiplos jogadores.
DirectShow	Permite que o *software* manipule facilmente a reprodução de vídeos, a partir de *Áudio Vídeo Interleaves* (AVIs) que gerenciam as placas de captura de vídeo.

Fonte: adaptado de Staroscik (2003).

Quadro 5 – Módulos do DirectX

2.2.1 DirectInput

O módulo `DirectInput` é o responsável pela comunicação dos aplicativos com os periféricos da máquina. Ele possui métodos de obtenção dos comandos de controladores de jogo (*device objects*), bem como o controle da interatividade da tecnologia *Force Feedback*, que nada mais é do que uma interface de interação háptica[6] com o jogador. De acordo com Microsoft (2009d), qualquer *hardware* que não seja categorizado como mouse ou teclado, é considerado um *joystick*.

O `DirectInput` trabalha diretamente com os *device drivers* dos dispositivos, criando uma nova camada de abstração entre o dispositivo e o aplicativo (conforme Figura 2) que recebe as entradas, sem a necessidade de saber que tipo de dispositivo está gerando-as. Antes do DirectX, o aplicativo devia prever uma lista de *joysticks* e compatibilizá-lo, de modo que

se comunicasse por si só com cada modelo e fabricante de *joystick*. Caso fosse lançado um modelo com novas funcionalidades que não estivessem previstas na implementação original do jogo, ele não sabia como se comunicar corretamente.

Sendo assim, com o DirectX fica completamente transparente para o desenvolvedor de que forma deve ser implementada a comunicação com um ou outro *joystick* plugado na máquina, bastando apenas programar o interfaceamento dos dados entre essa nova camada e seu aplicativo. Isto vem a se tornar extremamente útil quando se pensa na grande variedade de tipos e fabricantes de *joysticks* disponíveis no mercado (cada um com seu próprio conjunto de comandos e controle), pois alivia bastante a carga de responsabilidade do desenvolvedor na questão da compatibilidade com todos esses dispositivos em baixo nível (Figura 1).

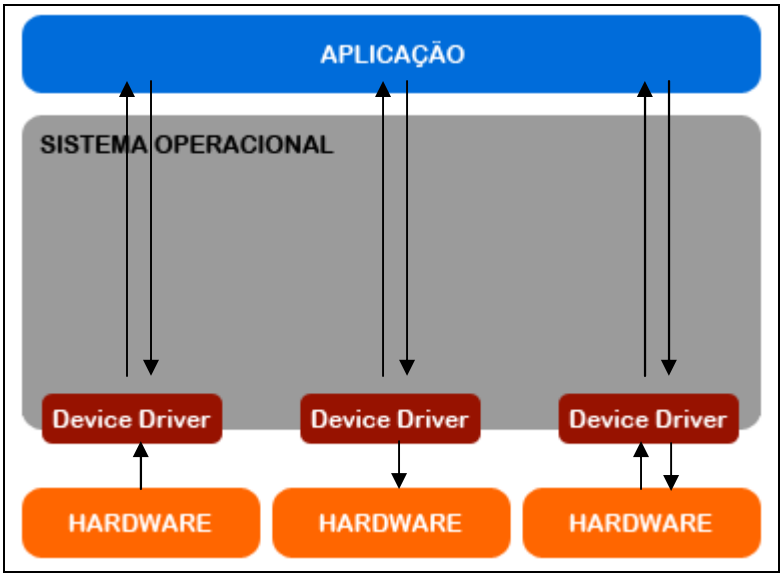


Figura 1 – Camadas de comunicação entre o aplicativo e o *hardware* (sem DirectX)

[6] Háptica: correlato tátil da óptica (para o visual) e da acústica (para o auditivo).

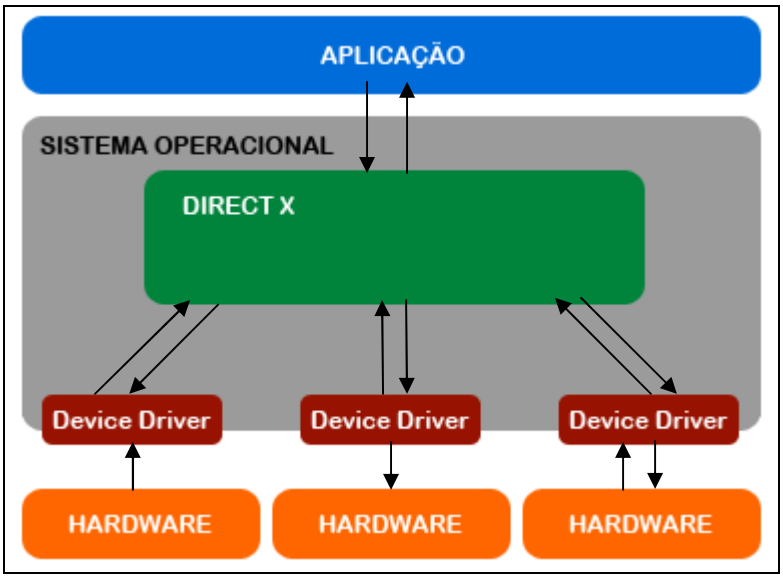


Figura 2 – Camadas de comunicação entre o aplicativo e o *hardware* (com DirectX)

Um ponto interessante a respeito desta camada é que por trabalhar diretamente com os *device drivers* dos dispositivos, as configurações do painel de controle do sistema operacional para o teclado e mouse são totalmente ignoradas. Sendo assim, a taxa de repetição de teclas e a inversão dos botões do mouse para destro/canhoto são exemplos de configurações que não são levadas em consideração durante a utilização destes dispositivos (MICROSOFT, 2009d). O que realmente interessa para o `DirectInput` é o estado das teclas (normal ou pressionada), e não a sua função.

O módulo `DirectInput` é constituído por 3 principais interfaces (`IDirectInput8`, `IDirectInputDevice8` e `IDirectInputEffect`) exportadas pela DLL `dinput8.dll`:

a) `IDirectInput8`: os métodos da interface `IDirectInput8` (Quadro 6) são utilizados para enumerar, criar e obter o status dos dispositivos, inicializar o objeto `DirectInput`, e invocar uma instância do painel de controle do Windows (MICROSOFT, 2009e);

Método	Descrição
`IDirectInput8::ConfigureDevices`	Mostra página de propriedades para os dispositivos de entrada conectados e permite que o usuário possa mapear ações para os controles do dispositivo.
`IDirectInput8::CreateDevice`	Cria e inicializa uma instância de um dispositivo baseando-se somente na GUID, e obtêm a interface `IDirectInputDevice8`.
`IDirectInput8::EnumDevices`	Enumera todos os dispositivos disponíveis.
`IDirectInput8::EnumDevicesBySemantics`	Enumera os dispositivos que mais se assemelham ao mapa de ações definido pelo aplicativo.
`IDirectInput8::FindDevice`	Retorna a GUID de um dispositivo que foi recém plugado ao sistema. Ele é chamado em resposta a uma notificação da administração de dispositivos.
`IDirectInput8::GetDeviceStatus`	Retorna o status de um determinado dispositivo.
`IDirectInput8::Initialize`	Inicializa um objeto `DirectInput`. Aplicativos normalmente não tem necessidade de chamar este método. A função `DirectInput8Create` automaticamente inicializa o objeto assim que ele é criado.
`IDirectInput8::RunControlPanel`	Executa o painel de controle, permitindo ao usuário instalar um novo dispositivo ou modificar configurações.

Fonte: Microsoft (2009e).

Quadro 6 – Métodos da interface `IDirectInput8`

b) `IDirectInputDevice8`: os métodos da interface `IDirectInputDevice8` (Quadro 7) são utilizados para obter e liberar acesso aos dispositivos, analisar suas propriedades e informações, ajustar comportamento, realizar a inicialização, criação e reprodução de efeitos *Force Feedback*, e invocar o painel de controle (MICROSOFT, 2009f);

Método	**Descrição**
`IDirectInputDevice8::Acquire`	Obtém acesso ao dispositivo de entrada.
`IDirectInputDevice8::BuildActionMap`	Cria um mapa de ações para o dispositivo e retorna informações sobre ele.
`IDirectInputDevice8::CreateEffect`	Cria e inicializa uma instância de um efeito identificado pelo GUID.
`IDirectInputDevice8::EnumCreatedEffectObjects`	Enumera todos os efeitos criados para o dispositivo.
`IDirectInputDevice8::EnumEffects`	Enumera todos os efeitos suportados pelo sistema *Force Feedback* do dispositivo.
`IDirectInputDevice8::EnumEffectsInFile`	Enumera todos os efeitos de um arquivo criado pelo utilitário Force Editor ou outro aplicativo que utiliza o mesmo formato.
`IDirectInputDevice8::EnumObjects`	Enumera os objetos de entrada e saída de dados disponíveis no dispositivo.
`IDirectInputDevice8::Escape`	Envia um comando específico de *hardware* para o *driver Force Feedback*.
`IDirectInputDevice8::GetCapabilities`	Obtém as capacidades do objeto `DirectInputDevice`.
`IDirectInputDevice8::GetDeviceData`	Retorna dados do *buffer* do dispositivo.
`IDirectInputDevice8::GetDeviceInfo`	Retorna informações sobre a identidade do dispositivo.
`IDirectInputDevice8::GetDeviceState`	Retorna informações imediatas a respeito do dispositivo.
`IDirectInputDevice8::GetEffectInfo`	Obtém informações a respeito de um efeito.
`IDirectInputDevice8::GetForceFeedbackState`	Retorna o estado do sistema *Force Feedback* do dispositivo.
`IDirectInputDevice8::GetImageInfo`	Retorna a imagem atual do dispositivo que pode ser utilizada para configuração de propriedades.
`IDirectInputDevice8::GetObjectInfo`	Retorna informações sobre um *device object*, como um botão ou um eixo.
`IDirectInputDevice8::GetProperty`	Retorna informações sobre o dispositivo de entrada.
`IDirectInputDevice8::Initialize`	Inicializa o objeto `DirectInputDevice`.
`IDirectInputDevice8::Poll`	Retorna dados sobre objetos pesquisados no dispositivo.
`IDirectInputDevice8::RunControlPanel`	Executa o painel de controle do `DirectInput` associado ao dispositivo.
`IDirectInputDevice8::SendDeviceData`	Envia dados para o dispositivo que aceita saída de dados.
`IDirectInputDevice8::SendForceFeedbackCommand`	Envia um comando para o sistema de *Force Feedback* do dispositivo.
`IDirectInputDevice8::SetActionMap`	Define o formado de dados para um dispositivo e mapeia ações definidas pelo aplicativo para os *device objects*.
`IDirectInputDevice8::SetCooperativeLevel`	Estabelece o nível de cooperação para a instância atual do dispositivo.
`IDirectInputDevice8::SetDataFormat`	Define o formato de dados para o dispositivo.
`IDirectInputDevice8::SetEventNotification`	Especifica um evento para ocorrer quando o status do dispositivo mudar.
`IDirectInputDevice8::SetProperty`	Estabelece propriedades que definem o comportamento do dispositivo.
`IDirectInputDevice8::Unacquire`	Libera o acesso ao dispositivo.

`IDirectInputDevice8::WriteEffectToFile`	Salva informações sobre um ou mais efeitos *Force Feedback* em um arquivo que pode ser lido usando o método `EnumEffectsInFile`

Fonte: Microsoft (2009f).

Quadro 7 – Métodos da interface `IDirectInputDevice8`

c) `IDirectInputEffect`: os métodos da interface `IDirectInputEffect` (Quadro 8) são utilizados para administrar efeitos de *Force Feedback* dos dispositivos (MICROSOFT, 2009g).

Método	**Descrição**
`IDirectInputEffect::Download`	Carrega o efeito no dispositivo.
`IDirectInputEffect::Escape`	Envia um comando específico de *hardware* para o *driver*.
`IDirectInputEffect::GetEffectGuid`	Obtém o GUID do efeito, representado pelo objeto `IDirectInputEffect`.
`IDirectInputEffect::GetEffectStatus`	Retorna o status de um efeito.
`IDirectInputEffect::GetParameters`	Obtém informações a respeito de um efeito.
`IDirectInputEffect::Initialize`	Inicializa um objeto `IDirectInputEffect`.
`IDirectInputEffect::SetParameters`	Parametriza as características de um efeito.
`IDirectInputEffect::Start`	Inicia a reprodução de um efeito.
`IDirectInputEffect::Stop`	Interrompe a reprodução de um efeito.
`IDirectInputEffect::Unload`	Remove o efeito do dispositivo.

Fonte: Microsoft (2009g).

Quadro 8 – Métodos da interface `IDirectInputEffect`

Microsoft (2009d) diz ainda que uma implementação apenas para leitura de dados, consiste em um objeto `DirectInput` (através da interface `IDirectInput8`), uma interface COM e um objeto `DirectInputDevice` para cada dispositivo que envia dados. Para cada `DirectInputDevice` há *device objects*, que nada mais são do que os botões, teclas ou eixos do dispositivo. Dispositivos compostos por diferentes meios de entrada de dados (como um teclado com *touchpad* embutido), têm a necessidade da criação de um `DirectInputDevice` para cada dispositivo de entrada, mesmo sendo constituído por um único *hardware*. Porém, um dispositivo com *Force Feedback* pode ser representado por um *joystick* apenas, que manipula entrada e saída de dados.

Assim que é instanciado um objeto `DirectInputDevice`, é necessário identificar a quantidade e o tipo de *device objects* que ele possui, através da função `EnumObjects`. Desta forma é possível saber o número e o tipo de botões ou teclas que o dispositivo oferece, quantos eixos direcionais, etc. sendo representados através de estruturas `DIDEVICEOBJECTINSTANCE` (MICROSOFT, 2009d).

2.3 DLL INJECTION

A DLL é uma fonte de código executável que reúne funções úteis para que programadores possam utilizar sem precisar reescrevê-las. O Sistema Operacional (SO) disponibiliza uma série de DLLs padrão, e basta ao desenvolvedor apenas referenciá-las para fazer uso de seus métodos.

De acordo com Palmenäs (2008), a técnica de DLL *injection* visa entrar no fluxo normal de execução de um *software* e introduzir uma rotina de código externa, como se fosse parte do aplicativo original. Este procedimento tem vários propósitos: desde adicionar novas funcionalidades ao aplicativo alvo, recolher e analisar dados do processamento, até a disseminação de *malwares*[7].

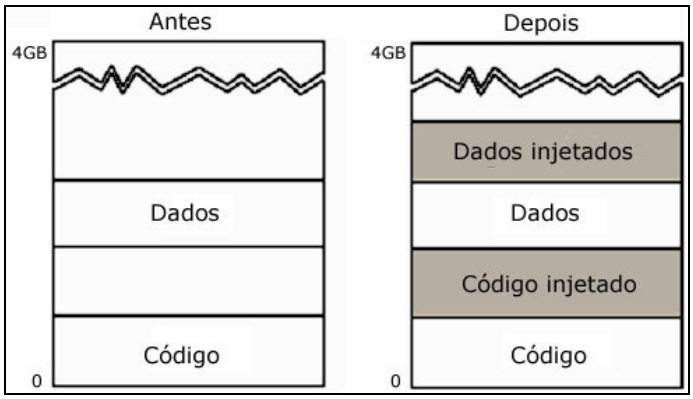
Uma vez carregada, a DLL injetada se comporta como qualquer outra DLL padrão chamada pelo aplicativo, pelo fato de suas rotinas de inicialização serem chamadas e de seus métodos exportados serem disponibilizados na tabela *Import Address Table* (IAT), que armazena ponteiros para os mesmos. Adicionalmente todas as dependências dessa DLL são buscadas (fornecendo todas as ferramentas comumente disponíveis a um executável), exatamente como é feito quando o aplicativo é iniciado (MILLER, TURKULAINEN, 2004, p. 4). Sendo assim, uma DLL injetada possui a mesma flexibilidade que se atribui a um aplicativo e ainda é capaz de ser executada dentro do contexto de um processo existente.

Para tanto, existem duas maneiras de injetar código:

a) DLL *injection*: uma DLL contendo o código a ser injetado é introduzida no meio do fluxo de processamento do aplicativo. Há mais de uma maneira de fazer isso, mas a mais fácil é adicionar a DLL a uma chave de registro do Windows que é responsável por listar todas as DLLs que são carregadas a cada execução de aplicativo no SO (PALMENÄS, 2008). Outra maneira é obter um identificador do processo alvo, alocar memória nele, e gravar a referência da DLL a ser carregada. É então criada uma nova *thread* para disparar a DLL, onde a API `LoadLibrary` chama o método `DllMain` que tem permissão para manipular o processo;
b) *Memory injection*: para Palmenäs (2008), trata-se de uma variante do método descrito acima, mas com a vantagem de não precisar carregar uma DLL separada e esperar que a `LoadLibrary` chame o método `DllMain`. Apenas um executável já é

[7] *Malware*: *software* que se infiltrar em um sistema e causa algum tipo de dano ou roubo de informações.

o suficiente para injetar código binário compilado para pronta execução pelo processo alvo. É necessário criar espaço na memória do aplicativo (Figura 3) através do método `WriteProcessMemory` e escrever o código a ser executado, e então uma nova *thread* é criada.



Fonte: adaptado de Palmenäs (2008).

Figura 3 – Organização da memória antes e depois de *Direct Memory Injection*

Diferentemente da execução de um aplicativo, alguns métodos de DLL *injection* não são externamente perceptíveis sem o uso de ferramentas especiais como o Process Explorer (Figura 4), que apresenta uma lista das DLLs carregadas em cada processo (MILLER, TURKULAINEN, 2004, p. 4).

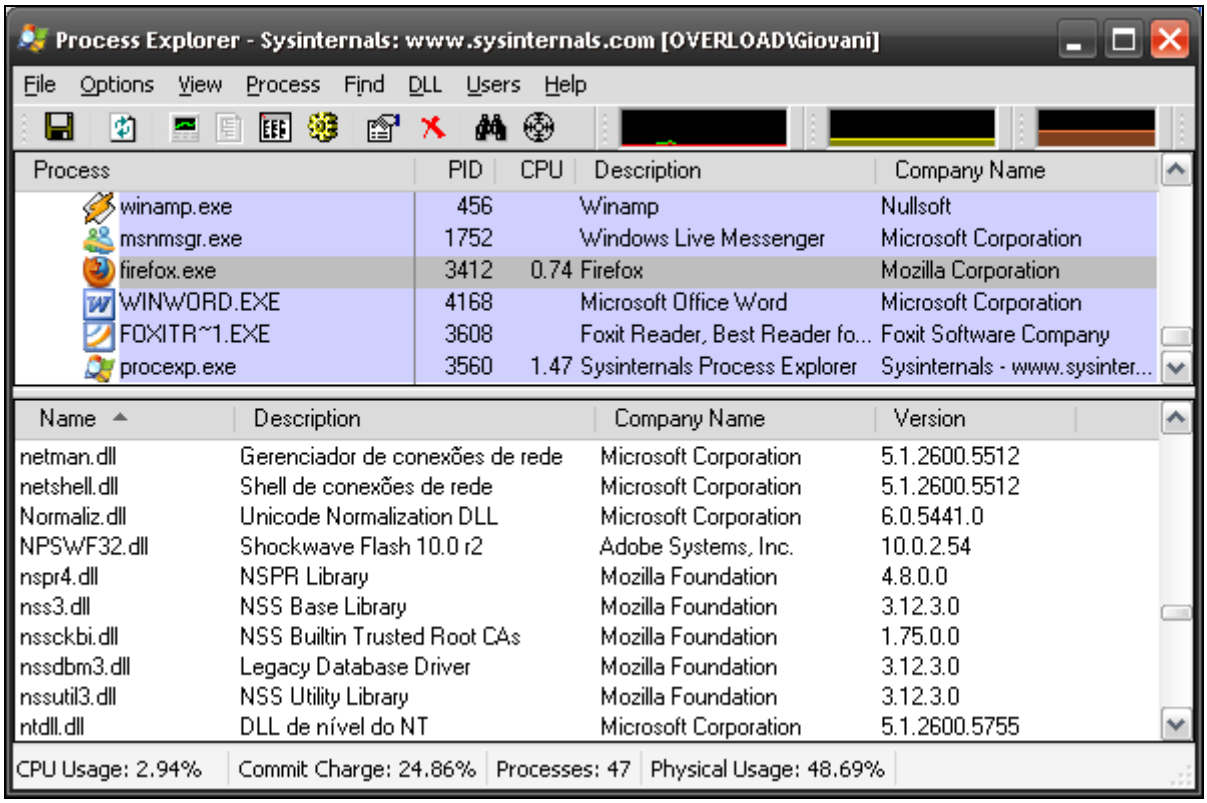


Figura 4 – Tela da ferramenta Process Explorer

2.4 HOOKS

"Tecnicamente, um *Hook* é somente uma outra subrotina (*Hook Procedure*) que fica no caminho do mecanismo normal de tratamento de mensagens do Windows" (SABAU, 2003). Esse tipo de implementação é bastante útil no caso de adicionar novas funcionalidades a aplicativos, também facilitando a comunicação entre os demais processos e mensagens do sistema. Os *Hooks* tendem a diminuir a performance do sistema por aumentar a carga de processamento que o sistema precisa executar para cada mensagem. Ele deve ser instalado somente quando necessário, e removido o mais cedo possível.

2.4.1 *Hook Chains*

Segundo Microsoft (2009b), o sistema permite muitos tipos diferentes de *Hooks*, cada tipo disponibilizando acesso a um aspecto diferente do seu mecanismo de manipulação. Desta forma, o sistema mantém um *Hook Chain* individual para cada tipo de *Hook*. Um *Hook Chain* é uma cadeia de ponteiros para funções especiais de *callback* de aplicativos. Quando ocorre uma mensagem que está associada a determinado tipo de *Hook*, o sistema passa essa mensagem para todas as subrotinas (*Hook Procedures*) desta *Hook Chain*, uma após a outra (Figura 5). A ação que cada *Hook Procedure* pode tomar depende do tipo de *Hook* envolvido. Algumas podem somente monitorar mensagens, outras podem modificar e até parar o progresso de mensagens pela cadeia, impedindo que atinja a próxima *Hook Procedure* ou até mesmo o seu destino final.

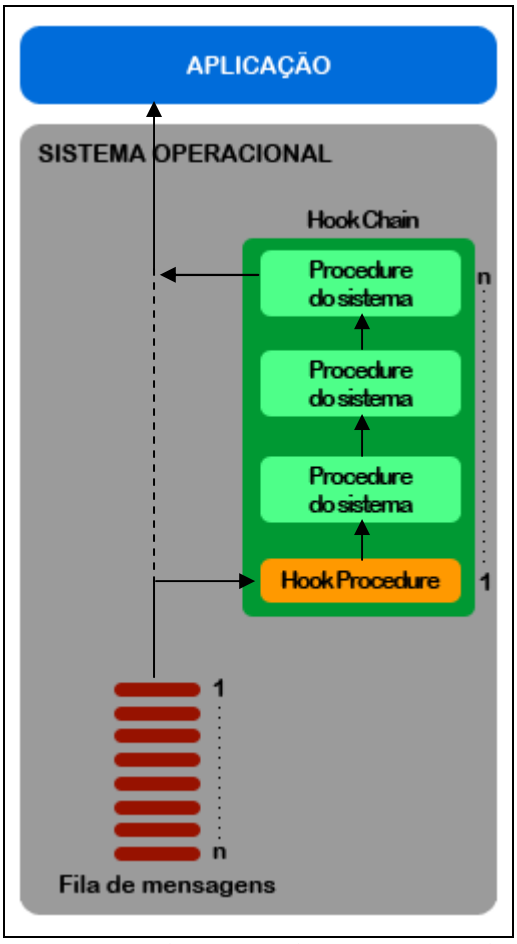


Figura 5 – Demonstração da captura de mensagens pela *Hook Chain*

2.4.2 *Hook Procedures*

Conforme Sabau (2003), "a *Hook Procedure* pode ser instalada no sistema e assim ela captura certas mensagens do Windows antes delas serem enviadas para as devidas rotinas de tratamento". Esse mecanismo é implementado através da chamada à função `SetWindowsHookEx` (Quadro 9), que instala a *procedure* referenciada em `lpfn` no *Hook Chain* especificado através de constantes de tipo no parâmetro `idHook`. A *procedure* precisa estar no mesmo processo do código referenciado pelo parâmetro `dwThreadId`. Caso não esteja, é necessária a implementação de uma DLL contendo a *procedure* a ser incluída externamente e referenciá-la no parâmetro `hMod`.

```
HHOOK SetWindowsHookEx(
    int idHook,
    HOOKPROC lpfn,
    HINSTANCE hMod,
    DWORD dwThreadId
);
```

Fonte: Microsoft (2009b).

Quadro 9 – Sintaxe de utilização do mecanismo de instalação de *Hook Procedures* à *Hook Chains*.

O Quadro 10 demonstra a sintaxe de declaração de uma *Hook Procedure* a ser anexada à *Hook Chain* especificada por parâmetro na função `SetWindowsHookEx()`.

```
LRESULT CALLBACK HookProc(
  int nCode,
  WPARAM wParam,
  LPARAM lParam
);
```

Fonte: Microsoft (2009c).

Quadro 10 – Sintaxe de declaração de *Hook Procedure*

A decisão sobre qual ação a *procedure* irá tomar quando for chamada é baseada no parâmetro `nCode`, que depende do tipo de *Hook* estabelecido. Cada tipo de *Hook* tem seus próprios códigos de *status* para identificar qual ação ocorreu, desde redimensionamento de janela até pressionamento de teclas. Os parâmetros `wParam` e `lParam` são coringas: dependendo do tipo de `nCode`, eles representam dados diferentes.

A *procedure* sempre é instalada no começo da *Hook Chain*. Quando ocorre uma mensagem, o sistema chama a *procedure* que está no começo da *Hook Chain* associada ao *Hook* do tipo da mensagem que ocorreu, e cada *procedure* tem o poder de decidir se vai ou não passá-la adiante. A função utilizada para chamar a próxima *procedure* da cadeia é `CallNextHookEx`. Porém, dependendo do tipo de *Hook*, a *procedure* somente poderá monitorar as mensagens, sendo que o sistema chamará a próxima *procedure* da cadeia tendo ou não a *procedure* atual chamado `CallNextHookEx` (MICROSOFT, 2009b).

> Existem os hooks de sistema (globais) que recebem mensagens de todos os threads do sistema, e os hooks específicos de thread (locais), que recebem mensagens apenas de um determinado thread. Devido a uma **hook procedure global** poder ser chamada no contexto de qualquer aplicação (que capturam mensagens de todas as aplicações), elas **devem estar localizadas em uma DLL** (Dynamic Link Library). Esta restrição não se aplica aos hooks específicos de threads, onde a hook procedure pode estar em qualquer parte da aplicação que controla o thread a ser interceptado. (SABAU, 2003, grifo nosso).

De acordo com Microsoft (2009b), os *Hooks* globais devem ser utilizados somente com propósitos de *debugging*, pois eles afetam criticamente o desempenho geral do sistema, e causam conflitos com outros aplicativos que implementam o mesmo tipo de *Hook* global; afinal, eles manipulam as mensagens de todas as *threads* do sistema.

2.4.3 Tipos de *Hook*

São apresentados abaixo os tipos de *Hook* segundo Microsoft (2009b), referenciados através das constantes:

a) `WH_CALLWNDPROC` e `WH_CALLWNDPROCRET` – monitoram as mensagens enviadas para as *procedures* da janela. `WH_CALLWNDPROC` é chamado antes de passar a mensagem para a *procedure* da janela que irá recebê-la, e `WH_CALLWNDPROCRET` é chamado após o processamento, passando um ponteiro para uma estrutura com os valores de retorno da *procedure* da janela;

b) `WH_CBT` – o sistema chama um *Hook* deste tipo antes de ativar, criar, destruir, minimizar, maximizar, mover ou redimensionar uma janela; antes de completar um comando do sistema, antes de remover um evento de mouse ou teclado da fila de mensagens do sistema, antes de definir o foco de entrada, ou antes de sincronizar com a fila de mensagens do sistema. O valor retornado pela *procedure* determina se o sistema permite ou impeça essas operações. Esse tipo de *Hook* foi implementado para aplicativos de treinamento baseados em computador;

c) `WH_DEBUG` – é chamado quando uma *Hook Procedure* de qualquer outro tipo de *Hook* é chamada, sendo útil para permitir ou não que o sistema chame-a;

d) `WH_FOREGROUNDIDLE` – este *Hook* permite que tarefas com baixa prioridade sejam executadas em *background* enquanto a *thread* principal está ociosa. É chamado quando a *thread* está para ficar ociosa;

e) `WH_GETMESSAGE` – monitora as mensagens que estão para ser retornadas pelas funções `GetMessage` ou `PeekMessage`. Também pode ser utilizado para monitorar as mensagens do mouse e teclado que enviadas para a fila de mensagens;

f) `WH_JOURNALRECORD` – permite que sejam monitorados e gravados os eventos de entrada. Tipicamente é utilizado para gravar uma sequência de mensagens do mouse e teclado e serem reproduzidos posteriormente através do *Hook* `WH_JOURNALPLAYBACK`. Este é um exemplo de *Hook* global, não podendo ser utilizado localmente pelo aplicativo;

g) `WH_JOURNALPLAYBACK` – permite que uma aplicação insira mensagens na fila de mensagens do sistema, como mensagens de mouse e teclado gravadas pelo `WH_JOURNALRECORD`. O mouse e teclado do sistema são desabilitados enquanto este *Hook* está instalado. Assim como o seu correspondente de captura, ele é global e

não pode ser instalado localmente;

h) `WH_KEYBOARD_LL` – monitora mensagens de teclado enviadas para a fila de entradas de uma *thread*;

i) `WH_KEYBOARD` – permite que um aplicativo monitore o tráfego de mensagens de `WM_KEYDOWN` e `WM_KEYUP` que estão para ser retornados pelas funções `GetMessage` ou `PeekMessage`, também podendo ser utilizado para o monitoramento de mensagens do teclado enviadas para a fila de mensagens do sistema;

j) `WH_MOUSE_LL` - monitora mensagens de mouse enviadas para a fila de entradas de uma *thread*;

k) `WH_MOUSE` - permite que um aplicativo monitore o tráfego de mensagens de mouse que estão para ser retornados pelas funções `GetMessage` ou `PeekMessage`, também podendo ser utilizado para o monitoramento de mensagens enviadas para a fila de mensagens do sistema;

l) `WH_MSGFILTER` ou `WH_SYSMSGFILTER` – monitoram as mensagens que estão para ser processadas por um menu, barra de rolagem, caixa de mensagem ou diálogo, e para detectar quando uma janela diferente está para ser ativada como resultado do pressionamento de uma combinação de teclas, como ALT+TAB ou ALT+ESC. O `WH_MSGFILTER` tem como escopo somente a aplicação local, e `WH_SYSMSGFILTER` abrange o sistema como um todo (global);

m) `WH_SHELL` – um aplicativo do tipo *shell* pode utilizá-lo para receber mensagens importantes. O sistema o chama quando a aplicação está para ser ativada, ou quando uma janela é criada ou destruída.

2.5 API HIJACK

Brainerd (2000) desenvolveu uma API (Quadro 11) para substituir funções de uma DLL carregada na memória por funções customizadas de uma DLL injetada no processo alvo através de um *Hook* global (do tipo WH_CBT, por exemplo).

```
SDLLHook D3DHook =
{
    "DDRAW.DLL",
    false, NULL,    // Default hook disabled, NULL function pointer.
    {
        { "DirectDrawCreate", MyDirectDrawCreate },
        { NULL, NULL }
    }
};

BOOL APIENTRY DllMain( HINSTANCE hModule, DWORD fdwReason, LPVOID
lpReserved)
{
    if ( fdwReason == DLL_PROCESS_ATTACH )  // When initializing....
    {
        hDLL = hModule;

        // We don't need thread notifications for what we're doing.  Thus,
        // get rid of them, thereby eliminating some of the overhead of
        // this DLL
        DisableThreadLibraryCalls( hModule );

        // Only hook the APIs if this is the right process.
        GetModuleFileName( GetModuleHandle( NULL ), Work, sizeof(Work) );
        PathStripPath( Work );

        if ( stricmp( Work, "myhooktarget.exe" ) == 0 )
            HookAPICalls( &D3DHook );
    }

    return TRUE;
}
```

Fonte: Brainerd (2000).

Quadro 11 – Código-fonte da APIHijack, desenvolvido por Wade Brainerd

O funcionamento desta API é bastante inteligente. É estabelecido um *Hook* do tipo `WH_CBT`, que carrega a DLL customizada na memória de todos os processos à medida que forem ocorrendo mensagens relacionadas à CBT. Assim que a DLL é carregada, o método `DllMain` é executado. O intuito do mecanismo CBT é monitorar todas as mensagens da janela para aplicações de treinamento baseado em computador; como não é esse o objetivo da API, essas mensagens são descartadas através do método `DisableThreadLibraryCalls()`, poupando recursos do sistema.

Logo a seguir, a DLL testa se o executável do processo ao qual ela está vinculada é o que foi escolhido para sofrer injeção, pois a DLL passa a ser carregada em todos os processos após a instalação do *Hook*. Desta forma, quando o executável escolhido é iniciado, a DLL é carregada na memória do processo e após se certificar de que está sendo processada no executável correto, ela chama o método `HookAPICalls()`.

Este método é responsável pela substituição das funções originais da DLL escolhida pelas funções customizadas da DLL injetada. Ele recebe como parâmetro uma estrutura

indicando a DLL e a função que será substituída, e a função customizada que será injetada. Apenas funções exportadas pela DLL selecionada podem ser substituídas. Essas funções podem ser visualizadas através da ferramenta Dependency Walker (Figura 6).

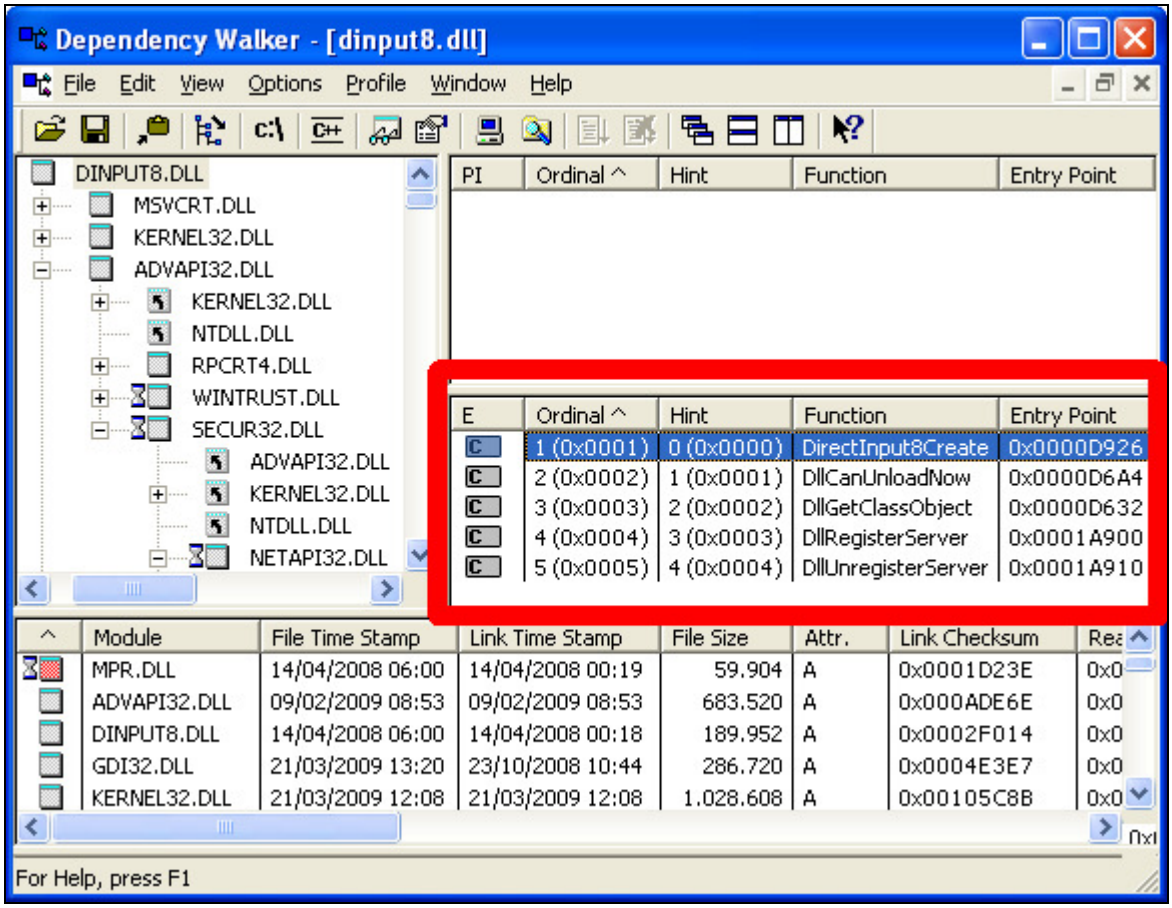


Figura 6 – Tela da ferramenta Dependency Walker, exibindo funções exportadas pela DLL

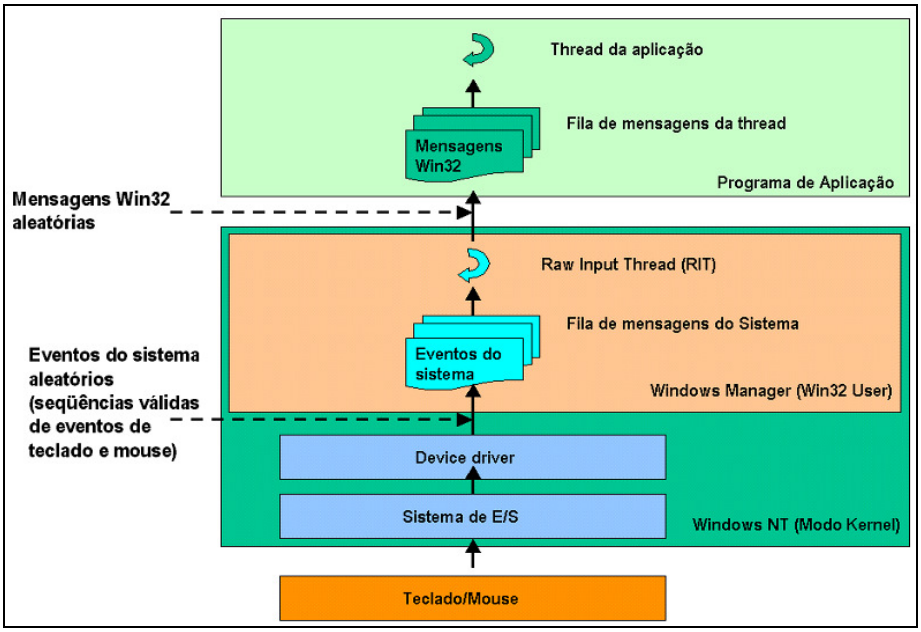
Cada processo no Windows tem uma tabela chamada *Import Address Table* (IAT), que armazena ponteiros para as funções exportadas pelas DLLs de cada processo. Essa tabela é preenchida dinamicamente com os endereços das funções das DLLs em tempo de execução. Por padrão, esta tabela é somente leitura. Assim que é chamado, o método `HookAPICalls()` torna a tabela IAT gravável através da função `VirtualProtect()`. Em seguida, ela itera por todos os métodos exportados por cada DLL do processo, procurando pela função que será substituída e troca o seu endereço pelo endereço da função customizada, voltando a marcar a tabela como somente leitura na sequência. A APIHijack adicionalmente salva uma cópia do ponteiro da função original para que ela possa ser utilizada na função customizada para retornar dados como se nada tivesse acontecido.

Quando o processo tenta chamar a função, o seu endereço é buscado na tabela IAT e um ponteiro é retornado. Como a tabela IAT foi modificada, a função customizada é chamada no lugar da função original e se obtém o código injetado no processo.

2.6 TRABALHOS CORRELATOS

Forrester e Barton (2000) descrevem os resultados dos testes de robustez em aplicações baseadas em Windows NT. A Figura 7 apresenta graficamente a estratégia usada para inserir eventos aleatórios nas filas de mensagens do Windows, de forma a testar as aplicações. Uma conclusão a que Forrester e Barton (2000) chegaram a partir dos testes realizados é que, "qualquer aplicação executando em plataformas Windows NT está vulnerável a sequências aleatórias de mensagens geradas por qualquer outra aplicação executando no mesmo sistema. Isto denota um aspecto crítico na arquitetura de mensagens do sistema".

Uma outra observação de Forrester e Barton (2000) é que os programadores também contribuem para que esta taxa de erros seja tão elevada. Segundo os autores, "parece haver um acordo entre os programadores para utilizar ponteiros diretamente a partir do conteúdo das mensagens sem antes realizar verificações de segurança". Esta situação conduz a uma infinidade de sub-versões e *patches* corretivos que já fazem parte do dia-a-dia da sociedade informatizada.



Fonte: adaptado de Forrester e Barton (2000).

Figura 7 – Mecanismo de inserção de mensagens aleatórias no Windows NT5.0

Em outro trabalho, Sun, Tseng e Lin (2006) discutem os métodos para infectar processos na plataforma Windows e descrevem um mecanismo denominado *Detecting the Code Injection Engine* (DCIE). Este *engine* é implementado como um *driver* em modo *kernel* e é capaz de detectar injeções de código em tempo de execução com *overhead* de 3,26%.

3 DESENVOLVIMENTO

Para detalhar o processo de desenvolvimento serão abordados os temas a seguir:

a) análise e especificação dos requisitos;
b) especificação através de diagramas de casos de uso, atividades e sequência;
c) estratégia para captura e interceptação das mensagens e principal técnica.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta deverá atender aos seguintes requisitos funcionais:

a) o aplicativo deverá interceptar, filtrar e identificar os eventos de *Force Feedback* do módulo `DirectInput`;
b) deverá apresentar uma lista dos *joysticks* instalados e permitir que o usuário selecione para qual deles os eventos serão redirecionados;
c) disponibilizar um campo de seleção de arquivo com filtro para somente executáveis, para que o usuário selecione o executável que sofrerá injeção;
d) o código injetado deverá permanecer ativo até que o usuário solicite seu desligamento através de um botão na tela.

A ferramenta deverá conter os seguintes requisitos não-funcionais:

a) o código do aplicativo de injeção deverá ser desenvolvido na linguagem C++, utilizando o ambiente de programação Microsoft Visual Studio 2008;
b) rodar sob sistema operacional Windows XP com DirectX 8 instalado ou superior;
c) utilizar o DirectX SDK para identificação dos métodos exportados pela DLL e importação dos tipos de dados.

3.2 ESPECIFICAÇÃO

Nesta seção são apresentados os diagramas de casos de uso, de classes, de atividades e de sequência. Os diagramas foram desenvolvidos seguindo a *Unified Modeling Language*

(UML) através da ferramenta Enterprise Architect versão 7.0 da empresa Sparx Systems.

3.2.1 Diagrama de casos de uso

O diagrama de casos de uso nesta seção tem como ator o usuário que fará a ativação do *Hook* no sistema para iniciar a captura e redirecionamento das mensagens DirectX. A Figura 8 descreve o usuário como ator que utilizará a única tela do aplicativo para ativar (Quadro 12) e desativar (Quadro 13) o *Hook*.

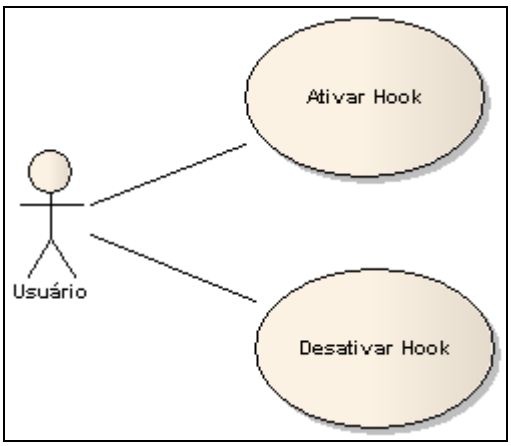


Figura 8 – Diagrama de casos de uso do ator usuário

Caso de uso	Ativar *Hook*.
Descrição	Instala no SO um *Hook* do tipo CBT para injetar a DLL customizada no executável especificado, com o intuito de redirecionar mensagens de Force Feedback da DirectX.
Ator	Usuário
Pré-condição	Haver pelo menos dois joysticks com Force Feedback instalados.
Cenário principal	Aplicativo apresenta uma lista com os joysticks instalados na máquina; Usuário seleciona um dos joysticks da lista para receber as mensagens replicadas; Usuário seleciona o executável que será injetado ou digita o seu nome; Usuário clica no botão para ativar o *hook*.
Cenário alternativo	No passo 1, caso não haja pelo menos 2 joysticks instalados no sistema, o aplicativo exibe uma mensagem e encerra. No passo 4, caso não tenha sido selecionado nenhum joystick e/ou executável, uma mensagem é exibida e volta ao passo 1.
Pós-condição	Eventos de Force Feedback são replicados no joystick selecionado, que fica oculto para o executável injetado.

Quadro 12 – Caso de uso Ativar *Hook*

Caso de uso	Desativar *Hook*.
Descrição	Desinstala do SO o *Hook* instalado previamente.
Ator	Usuário
Pré-condição	*Hook* estar instalado no SO. Aplicativo injetado estar finalizado.
Cenário principal	Usuário clica no botão para desativar *Hook*; *Hook* é desinstalado do SO.
Cenário alternativo	No passo 2, caso o aplicativo não tenha sido finalizado, ocorre um erro fatal pois as referências de memória dos métodos apontam para uma área que não existe mais.
Pós-condição	*Hook* é desinstalado do SO.

Quadro 13 – Caso de uso Desativar *Hook*

3.2.2 Diagrama de classes

O diagrama de classes apresentado abaixo (Figura 9) reflete os métodos exportados pela `dinput8.dll` da DirectX, que é a DLL a ser substituída. Contém as três principais interfaces que fazem o controle dos *joysticks*, tendo alguns métodos modificados com o intuito de replicar os eventos de *Force Feedback* para outro *joystick* e escondê-lo da aplicação para evitar conflitos (um *joystick* enviando eventos para ele mesmo).

Figura 9 – Interfaces do módulo `DirectInput`

3.2.3 Diagrama de atividades

O diagrama de atividades (Figura 10 e Figura 11) é um diagrama que representa os fluxos conduzidos por processamentos. É essencialmente um fluxograma, mostrando o fluxo de controle de uma atividade para outra. Comumente isso envolve a modelagem das etapas sequenciais em um processamento computacional.

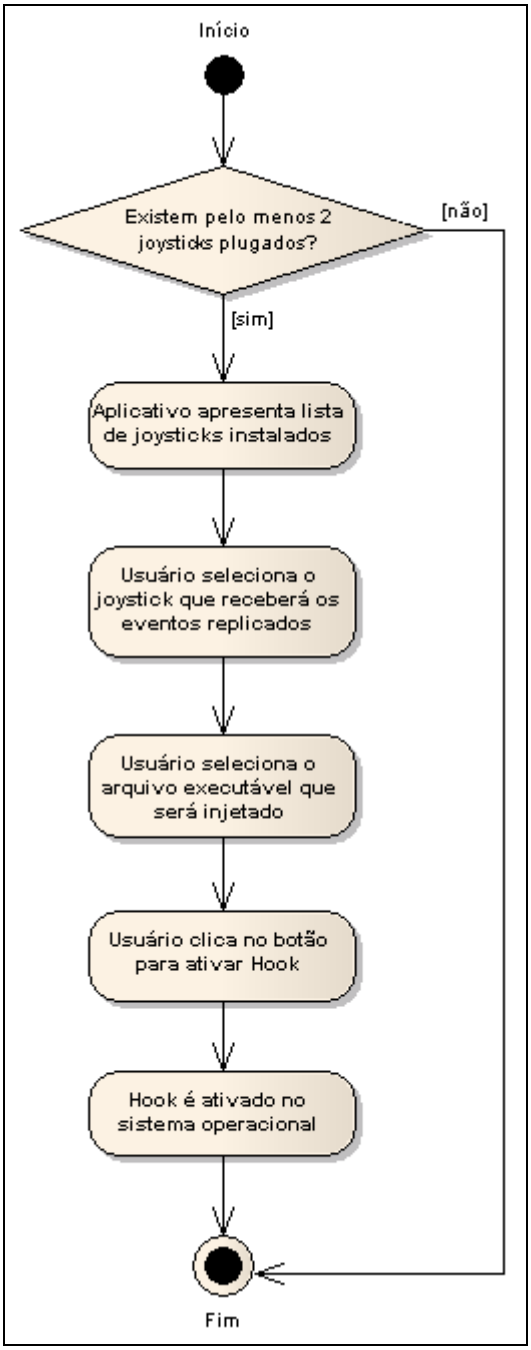


Figura 10 – Diagrama de atividades do aplicativo

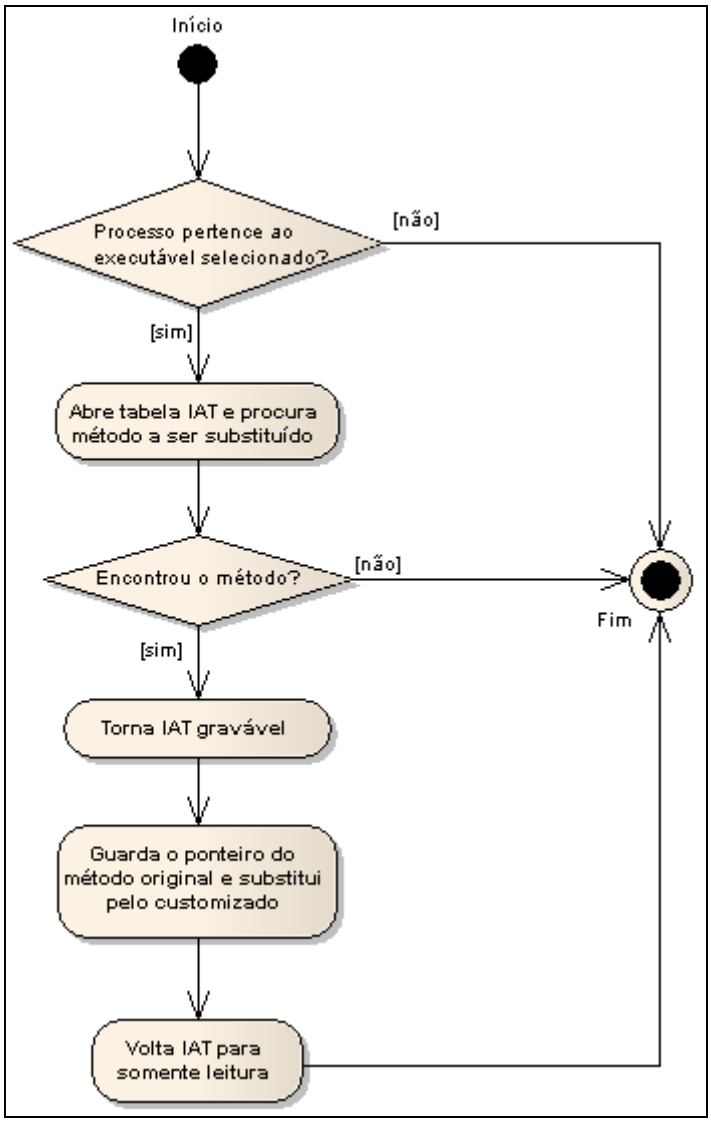


Figura 11 – Diagrama de atividades da DLL

3.2.4 Diagrama de sequência

No diagrama de sequência apresentado na Figura 12 é especificado o funcionamento do processo de injeção da DLL customizada no processo do executável selecionado. A DLL é carregada na memória, e a cada mensagem CBT que ocorrer no SO, a DLL é chamada tendo o método `DllMain` executado. Ele compara o nome do executável do processo envolvido na mensagem com o nome do executável selecionado, e caso sejam iguais, é iniciado o procedimento de substituição do método exportado pela DLL original pelo método customizado da DLL injetada. Adicionalmente, é guardado o ponteiro do método original para uso posterior.

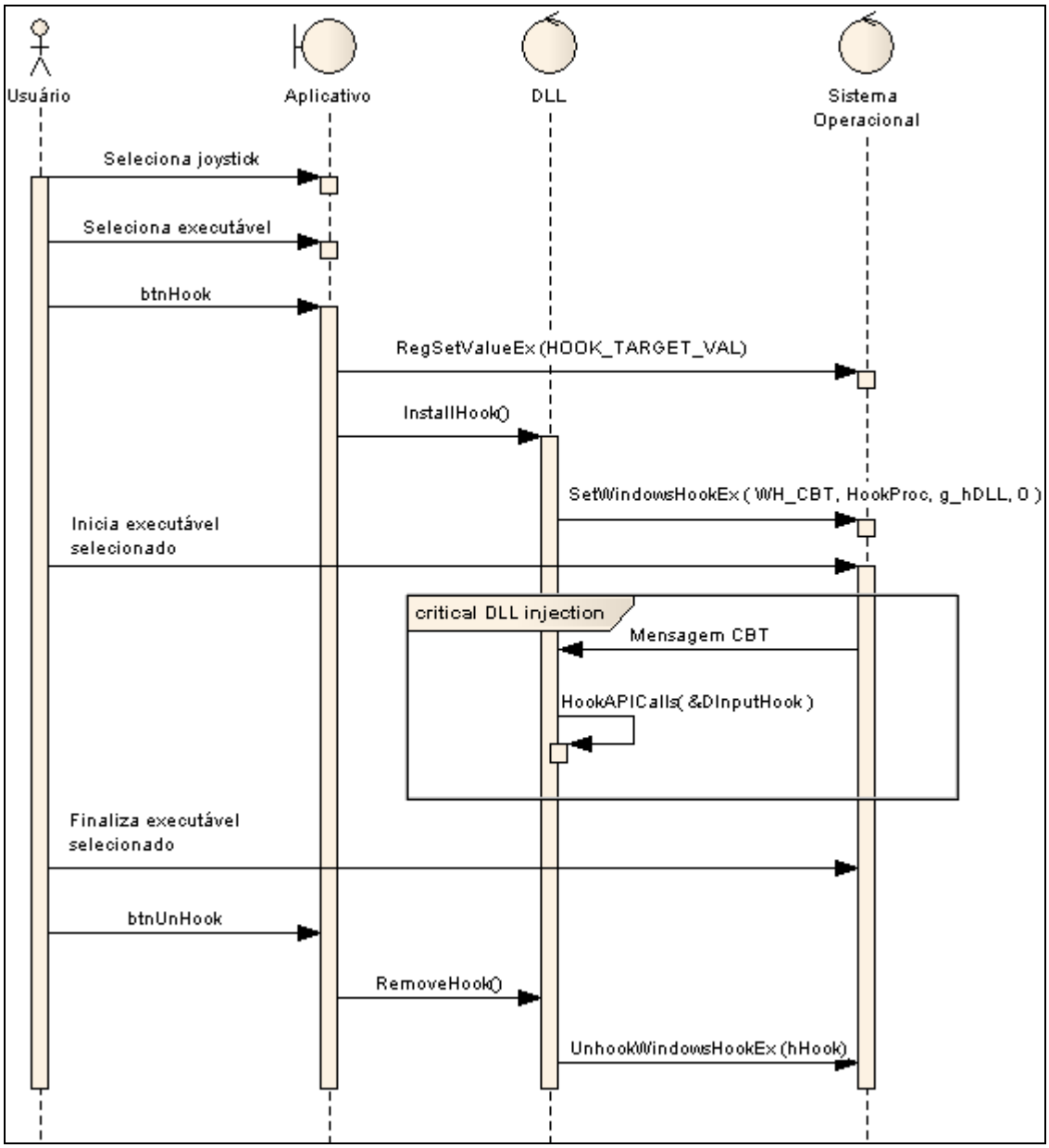


Figura 12 – Diagrama de sequência

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação do aplicativo foi utilizada a linguagem de programação C++, por oferecer facilidades na manipulação de ponteiros, e o DirectX SDK para obtenção de todos os métodos das interfaces injetadas e os tipos de dados. O ambiente de desenvolvimento escolhido foi o Microsoft Visual Studio 2008, por ser um produto do mesmo fabricante do SDK utilizado no desenvolvimento e compatível com o mesmo. O ambiente deve ser integrado com o SDK a fim de possibilitar a importação dos arquivos contendo as constantes de tipo utilizadas no código-fonte.

3.3.2 Aplicativo

O aplicativo desenvolvido (Figura 13) apresenta uma interface que requer apenas alguns dados para executar sua função de ativar o *Hook*. Ele apresenta uma lista com os *joysticks* instalados no sistema que são compatíveis com a tecnologia *Force Feedback* retornados pela própria DirectX, garantido assim a compatibilidade com o projeto (Quadro 14). Também oferece um campo para que o usuário especifique o nome do executável que sofrerá a injeção, com um botão de seletor de arquivos.

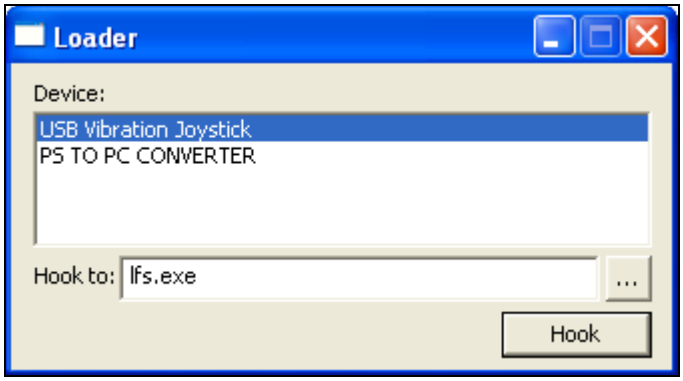


Figura 13 – Tela do aplicativo

```
void CDlgData::LoadControllers()
{
      m_Controllers.clear();

      IDirectInput8* pDI;
      HRESULT hr = DirectInput8Create( GetModuleHandle( NULL ),
DIRECTINPUT_VERSION, IID_IDirectInput8, (void**)&pDI, NULL );

      if( SUCCEEDED( hr ) )
      {
            pDI->EnumDevices( DI8DEVCLASS_GAMECTRL,
&EnumControllersCBStatic, this, DIEDFL_FORCEFEEDBACK | DIEDFL_ATTACHEDONLY
);
            pDI->Release();
      }
}
```

Quadro 14 – Método `LoadControllers()`

Assim que os dados são preenchidos e o usuário clica no botão "Hook", as informações sobre qual *joystick* foi selecionado (GUID do dispositivo) e o nome do executável são gravados no registro do SO, pois não é possível passar esses parâmetros diretamente para a *Hook Procedure* da DLL que será injetada. Assim que é clicado, o mesmo botão tem sua função modificada e passa a desativar o *Hook*.

Para desativar o *Hook*, o usuário deverá antes se certificar de que o executável injetado já tenha sido finalizado. Caso o *Hook* seja desativado enquanto o executável ainda está sendo processado, os ponteiros de memória referenciados na tabela IAT continuarão apontando para uma área de memória que não existe mais (no caso, do método customizado da DLL da *Hook Procedure* desativada), causando um erro fatal.

3.3.3 DLL *injection*

Há diversas maneiras de efetuar o processo de DLL *injection* em um processo ativo. Palmenäs (2008) descreve pelo menos duas maneiras de realizar o procedimento, mas uma terceira maneira foi utilizada: o estabelecimento de um *Hook* global para carregar a DLL no processo.

Durante a manipulação de janelas no SO, diversos eventos são disparados através de mensagens que passam por suas devidas *Hook Chains*. As mensagens das ações mais comuns que são realizadas durante a operação da maioria dos aplicativos (como ativar, criar, destruir, minimizar, maximizar, mover ou redimensionar janelas) passam pela *Hook Chain* do tipo WH_CBT. A ativação do *Hook* é feita através da função `SetWindowsHook()`, conforme

Quadro 15, apenas sendo especificado o tipo de *Hook*, a *Hook Procedure* criada para efetuar a injeção do código no processo, e a referência da DLL que contém a *Hook Procedure*. O último parâmetro deve ser definido como "0" pois a *Hook Procedure* está sendo buscada em uma DLL, e não em uma *thread*.

```
#pragma data_seg( ".HookSection" )
HHOOK hHook = NULL;     // Shared instance for all processes.
#pragma data_seg()

DIHOOK_API LRESULT CALLBACK HookProc( int nCode, WPARAM wParam, LPARAM
lParam )
{
    return CallNextHookEx( hHook, nCode, wParam, lParam);
}

extern "C"
{
      DIHOOK_API void __stdcall InstallHook()
      {
            hHook = SetWindowsHookEx( WH_CBT, HookProc, g_hDLL, 0 );
      }

      DIHOOK_API void __stdcall RemoveHook()
      {
            UnhookWindowsHookEx( hHook );
      }
}
```

Quadro 15 – Métodos de ativação e desativação do *Hook*

Um ponto importante que deve ser notado, é que a *procedure* `HookProc` apenas chama a próxima *Hook Procedure* através da função `CallNextHookEx()`, sem fazer qualquer outra manipulação ou chamada de método. Isso se dá por consequência da implementação ter sido feita dentro do método `DllMain`, que é chamado automaticamente assim que a DLL é carregada no processo.

3.3.4 Método `DllMain`

Uma vez que a DLL já está carregada na memória e referenciada através do *Hook* global WH_CBT, ela passa a ser carregada na memória de todos os processos gradativamente à medida que forem ocorrendo ações que geram mensagens que passam pela sua *Hook Chain*. No momento em que ela é carregada em um processo, o método `DllMain` (equivalente ao `void main` do C++) é executado automaticamente (Quadro 16).

Pelo fato da DLL ser carregada na memória de todos os processos ativos, torna-se necessário verificar o nome do executável do processo e compará-lo ao nome do executável

selecionado para injeção, que foi previamente recuperado do registro do SO.

```
BOOL APIENTRY DllMain( HMODULE hModule, DWORD dwReason, void* lpReserved )
{
	if( dwReason == DLL_PROCESS_ATTACH )
	{
		g_hDLL = hModule;

		HKEY hAppKey;

		if( RegOpenKeyEx( HKEY_CURRENT_USER, HOOK_REG_KEY, 0,
KEY_QUERY_VALUE, &hAppKey ) == ERROR_SUCCESS )
		{
			TCHAR szTargetProcess[512];
			DWORD dwCount = sizeof( szTargetProcess );

			if( RegQueryValueEx( hAppKey, HOOK_TARGET_VAL, NULL,
NULL, (BYTE*)szTargetProcess, &dwCount ) == ERROR_SUCCESS )
			{
				TCHAR szProcess[512];
				GetModuleFileName( GetModuleHandle( NULL ),
szProcess, _countof( szProcess ) );
				PathStripPath( szProcess );

				if( _strnicmp( szTargetProcess, szProcess,
_countof( szProcess ) ) == 0 &&
					LoadSeatGuid( hAppKey, &g_guidSeat ) )
				{
					FazLog( "Initializing hook\r\n" );
					HookAPICalls( &DInputHook );
				}
			}

			RegCloseKey( hAppKey );
		}
	}
	else if( dwReason == DLL_PROCESS_DETACH )
	{
		if( g_hLogFile != NULL )
		{
			CloseHandle( g_hLogFile );
			g_hLogFile = NULL;
		}
	}

	return TRUE;
}
```

Fonte: Haggag (2005).

Quadro 16 – Método `DllMain`

Caso o processo em que a DLL está carregada for do executável previamente selecionado, o método `HookAPICalls()` fornecido pela APIHijack é chamado e passado uma estrutura contendo os parâmetros que identificam quais métodos devem ser substituídos e quais devem substituir, conforme Quadro 17.

```
SDLLHook DInputHook =
{
      "DINPUT8.DLL", false, NULL,         // Default hook disabled, NULL
function pointer.
      {
            { "DirectInput8Create", &MyDirectInput8Create },
            { NULL, NULL }
      }
};

// Hook function.
HRESULT WINAPI MyDirectInput8Create( HINSTANCE hinst, DWORD dwVersion,
REFIID riidltf, void** ppvOut, LPUNKNOWN punkOuter )
{
      PFN_DirectInput8Create pfnOldFunc =
static_cast<PFN_DirectInput8Create>( DInputHook.Functions[0].OrigFn );
      HRESULT hr = pfnOldFunc( hinst, dwVersion, riidltf, ppvOut, punkOuter
);

      if( SUCCEEDED( hr ) )
      {
            *ppvOut = new MyDirectInput8( reinterpret_cast<IDirectInput8*>(
*ppvOut ) );
      }

      return hr;
}
```

Quadro 17 – Estrutura e função substitutiva

3.3.5 APIHijack

A APIHijack, desenvolvida por Brainerd (2000), é o coração desta implementação. Nela, através da função `HookAPICalls()` todos os descritores de importação (que relacionam os métodos exportados para cada DLL) são iterados até que seja encontrada a DLL especificada na estrutura `SDLLHook` (Quadro 18). Assim que é encontrada, a função `RedirectIAT()` é chamada. Ela torna a tabela IAT, que é somente leitura, gravável (conforme Quadro 19), e todos os métodos exportados pela DLL especificada na estrutura `SDLLHook` são iterados. A cada iteração, a função também itera a lista de métodos a serem substituídos, passada pela estrutura `SDLLHook` (Quadro 20).

Quando um método a ser substituído é encontrado, a APIHijack extrai da tabela IAT a referência de memória do ponteiro para o método e salva. Desta forma, é possível efetuar um *bypass* nas funções em que não é necessário efetuar nenhuma manipulação, apenas referenciando a mesma função do método original, conforme Quadro 21.

Logo após o método ser substituído, as permissões da tabela IAT são restauradas como se nada tivesse acontecido.

```
bool HookAPICalls( SDLLHook* Hook )
{
    if ( !Hook )
        return false;

    HMODULE hModEXE = GetModuleHandle( 0 );

    PIMAGE_NT_HEADERS pExeNTHdr = PEHeaderFromHModule( hModEXE );

    if ( !pExeNTHdr )
        return false;

    DWORD importRVA = pExeNTHdr->OptionalHeader.DataDirectory
                        [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress;
    if ( !importRVA )
        return false;

    // Convert imports RVA to a usable pointer
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc = MakePtr(
PIMAGE_IMPORT_DESCRIPTOR,
                                                    hModEXE, importRVA );

    // Save off imports address in a global for later use
    g_pFirstImportDesc = pImportDesc;

    // Iterate through each import descriptor, and redirect if appropriate
    while ( pImportDesc->FirstThunk )
    {
        PSTR pszImportModuleName = MakePtr( PSTR, hModEXE, pImportDesc-
>Name);

        if ( lstrcmpi( pszImportModuleName, Hook->Name ) == 0 )
        {
            OutputDebugString( "Found " );
            OutputDebugString( Hook->Name );
            OutputDebugString( "...\n" );

            RedirectIAT( Hook, pImportDesc, (PVOID)hModEXE );
        }

        pImportDesc++;  // Advance to next import descriptor
    }

    return true;
}
```

Quadro 18 – Método `HookAPICalls()`

```
...
    // These next few lines ensure that we'll be able to modify the IAT,
    // which is often in a read-only section in the EXE.
    DWORD flOldProtect, flNewProtect, flDontCare;
    MEMORY_BASIC_INFORMATION mbi;

    // Get the current protection attributes
    VirtualQuery( pIAT, &mbi, sizeof(mbi) );

    // remove ReadOnly and ExecuteRead attributes, add on ReadWrite flag
    flNewProtect = mbi.Protect;
    flNewProtect &= ~(PAGE_READONLY | PAGE_EXECUTE_READ);
    flNewProtect |= (PAGE_READWRITE);

    if ( !VirtualProtect(   pIAT, sizeof(PVOID) * cFuncs,
                            flNewProtect, &flOldProtect) )
    {
        return false;
    }
...
```

Quadro 19 – Fragmento do método `RedirectIAT()`

```
...

    pIteratingIAT = pIAT;

    while ( pIteratingIAT->u1.Function )
    {
        void* HookFn = 0;  // Set to either the SFunctionHook or pStubs.

        if ( !IMAGE_SNAP_BY_ORDINAL( pINT->u1.Ordinal ) )  // import by
name
        {
            PIMAGE_IMPORT_BY_NAME pImportName = MakePtr(
PIMAGE_IMPORT_BY_NAME, pBaseLoadAddr, pINT->u1.AddressOfData );

            // Iterate through the hook functions, searching for this
import.
            SFunctionHook* FHook = DLLHook->Functions;
            while ( FHook->Name )
            {
                if ( lstrcmpi( FHook->Name, (char*)pImportName->Name ) ==
0 )
                {
                    OutputDebugString( "Hooked function: " );
                    OutputDebugString( (char*)pImportName->Name );
                    OutputDebugString( "\n" );

                    // Save the old function in the SFunctionHook
structure and get the new one.
                    FHook->OrigFn = reinterpret_cast<void*>(pIteratingIAT-
>u1.Function);
                    HookFn = FHook->HookFn;
                    break;
                }
...
```

Quadro 20 – Fragmento do método `RedirectIAT()`

```
...

    STDMETHOD(EnumDevices)( DWORD devType,LPDIENUMDEVICESCALLBACK
callback, LPVOID ref, DWORD flags )
      {
            FazLog( "Begin enum\r\n" );

            MyCallbackContext myContext;
            myContext.pOriginalCallback = callback;
            myContext.pOriginalContext = ref;

            return m_dip->EnumDevices( devType, &MyEnumCallback,
&myContext, flags );
      }

    STDMETHOD(GetDeviceStatus)( REFGUID rguid )
      {
            return m_dip->GetDeviceStatus( rguid );
      }

    STDMETHOD(RunControlPanel)( HWND owner, DWORD flags )
      {
            return m_dip->RunControlPanel( owner, flags );
      }
...
```

Quadro 21 – Fragmento da interface `DirectInput`

3.3.6 Operacionalidade da implementação

Para caracterizar a operacionalidade da implementação, foi utilizado o seguinte cenário (Figura 14):

a) Um jogo com suporte a *Force Feedback* chamado Live for Speed;

b) Dois *joysticks* USB de marcas genéricas com suporte a *Force Feedback*.

Figura 14 – Cenário de testes

Num primeiro momento o jogo é executado com os dois *joysticks* plugados, porém apenas o primeiro recebe os eventos de *Force Feedback* desencadeados pelas ações no decorrer do jogo, sendo que o segundo permanece inerte durante todo o tempo. O jogo não suporta dois *joysticks* recebendo *Force Feedback* ao mesmo tempo. Depois de instalada a DLL desenvolvida (Figura 15), o efeito percebido é de que os eventos de *Force Feedback* são replicados nos dois *joysticks* simultaneamente.

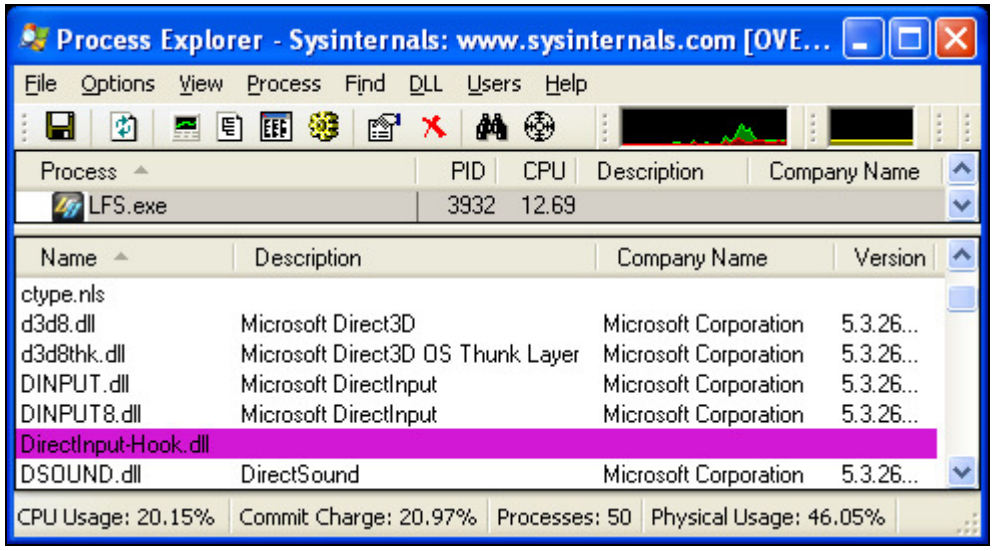


Figura 15 – DLL injetada no processo do executável LFS.exe

Para demonstrar a funcionalidade, selecionou-se o jogo Live for Speed (Figura 16) por ser um simulador de corridas 3D realístico *multiplayer online*, que também permite corridas

locais com apenas um jogador contra outros carros controlados pela inteligência artificial do aplicativo. Ele oferece suporte a volantes e *joysticks*, inclusive com a tecnologia *Force Feedback*. A distribuição é feita por *download* pelo próprio site do fabricante, sendo necessário licenciá-lo para desbloquear todos os recursos. Na versão bloqueada, disponibiliza apenas uma pista e três carros e não tem limite de tempo de expiração.

Fonte: Live for Speed (2009).

Figura 16 – Tela do jogo Live for Speed

A ferramenta (Figura 13) apresenta as seguintes funcionalidades verificadas na seção 3.1, descritas como segue:

a) selecionar *joystick*: é apresentada uma lista com todos os *joysticks* instalados na máquina, e o usuário deverá selecionar um deles para receber as mensagens redirecionadas. São necessários pelo menos dois *joysticks*, e o *joystick* selecionado não estará disponível para o aplicativo que sofrerá injeção;
b) especificar executável: é possível selecionar o nome do arquivo executável que sofrerá injeção de duas formas. Uma é digitando o nome diretamente no campo de texto, outra é selecionando através de uma caixa para seleção de arquivos com filtro para somente executáveis;
c) ativar *Hook*: após especificados o *joystick* e o executável, o aplicativo está pronto para ativar o *Hook* que fará a DLL *injection*.

3.4 RESULTADOS E DISCUSSÃO

O processo de DLL *injection* é executado com êxito pelo aplicativo, carregando as funções da DLL injetada no processo e substituindo na IAT as funções da DLL escolhida. Porém, devido à abordagem adotada para atingir este objetivo (*Hook* `WH_CBT`), a DLL também é carregada em todos os demais processos que estão sendo executados à medida que geram mensagens CBT. Apesar de haver uma checagem para que não seja feita qualquer alteração nos demais processos, recursos são consumidos desnecessariamente.

Uma solução seria utilizar a técnica descrita por Palmenäs (2008), de injetar a DLL diretamente na memória do processo. Através da função `VirtualAllocEx`, uma área de memória é alocada no processo para a gravação do endereço da DLL fornecido pela função `GetModuleFileName`., sendo posteriormente carregada pela função `LoadLibrary`. Logo em seguida uma nova *thread* é criada através da função `CreateRemoreThread` para inicializar a DLL.

Porém, dependendo de como o executável injetado foi implementado e das funções que serão substituídas, essa abordagem pode não funcionar como esperado. Caso a função que será substituída já tenha sido chamada pelo executável antes da injeção ocorrer, o procedimento não funcionará. Por isso, a injeção deve ocorrer tão logo o aplicativo seja inicializado, para que assim que a função seja chamada, ela já tenha sido substituída e efetue o processamento customizado.

Um aspecto importante a destacar é que o projeto ampliou os objetivos iniciais à medida em que foi possível demonstrar uma aplicação prática da adoção da tecnologia, utilizando-se um assento massageador que recebe estímulos de *Force Feedback* durante a execução de jogos de computador. Um produto comercial com a mesma funcionalidade é apresentado em Parisi (2009), denominado Car Massager (Figura 17).

Figura 17 – Acessório desenvolvido para simular o assento de um piloto de corrida

Quanto ao interfaceamento do assento com o computador, o fabricante Delcom disponibiliza chips USB de categoria 03h (*Human Interface Device*) e projetos montados de comunicação externa para uso genérico incluindo *device drivers*. Porém, além da necessidade de desenvolvimento de uma placa eletrônica, seu custo elevado inviabiliza a implementação desta forma.

Percebeu-se então que é possível utilizar um *joystick* USB com *Force Feedback* (como um dos utilizados no cenário de testes – Figura 14), que já tem placa eletrônica e seus *device drivers* previamente desenvolvidos pelo próprio fabricante, custando menos de 20% do produto da Delcom. Efetuando-se a adaptação do *hardware* de comando dos motores atuadores que controlam a vibração do *joystick* aos motores do assento massageador, foi obtida a comunicação entre o computador e o assento (Figura 18). Como o *joystick* adaptado não é mais utilizado como forma de entrada de dados mas apenas de saída, os botões não foram mais necessários e puderam ser removidos.

Figura 18 – Adaptação de *joystick* USB ao assento massageador Car Massager

Relativamente aos trabalhos correlatos, deve-se considerar a similaridade do uso da técnica de injeção e *Hooking* porém com propósitos diferentes. Como visto, Forrester e Barton (2000) utilizaram a estratégia para inserir eventos aleatórios nas filas de mensagens do Windows e Sun, Tseng e Lin (2006) utilizaram a tecnologia para detectar injeção de código em tempo de execução. O presente trabalho utilizou a tecnologia para redirecionar mensagens filtradas.

4 CONCLUSÕES

Conforme o estudo realizado sobre DLL *injection*, fica bastante claro que aberturas deste tipo nos *softwares* podem ser encaradas como brechas de segurança em aplicativos, pela facilidade com que são efetuadas. Os dados precisam estar protegidos de qualquer modificação externa, para evitar corrupção de informações, *malwares* ou até mesmo ataques *hacker*.

Por outro lado, este tipo de abertura de acesso aos dados proporciona, além de implementações de extensões em aplicativos, o desenvolvimento de novos *softwares* que precisam de certo grau de comunicação com os demais processos em execução.

Foram utilizados *Hooks* como forma de carregar a DLL no processo do executável selecionado ao invés dos métodos propostos por Palmenäs por ser mais simples de implementar e ter maior compatibilidade. Adicionar uma entrada no registro para carregar a DLL customizada poderia ser encarado pelo SO como uma tentativa de ataque por *malware*, e abrir área de memória no aplicativo já executando inviabilizaria a implementação, pois no caso do Live for Speed é feito o reconhecimento dos *joysticks* instalados na máquina dentro das suas rotinas de inicialização; além de ser possível o desencadeamento de um erro de violação de acesso, pois varia muito a forma como cada executável de jogo é implementado.

Uma das dificuldades encontradas foi a necessidade de tornar o *joystick* selecionado indisponível para o aplicativo selecionado, uma vez que os eventos serão redirecionados para ele mesmo caso seja utilizado no jogo. Assim, foi necessário manipular também a função de *callback* que enumera e retorna para o aplicativo todos os *joysticks* instalados na máquina, para que o *joystick* selecionado seja escondido.

Durante todo o desenvolvimento do trabalho, notou-se rica documentação sobre DirectX no site do fabricante, além da grande quantidade de funções variadas que são disponibilizadas pela API, tornando mais fácil e rápido o desenvolvimento de aplicações que a utilizam.

4.1 EXTENSÕES

Como extensão para esse trabalho, sugere-se a adaptação de um *joystick* com suporte a

Force Feedback a um dispositivo vibratório de forma que transfira também para o corpo do jogador todas as sensações que um piloto de verdade sente enquanto está pilotando. Dado o grau de realismo apresentado pelo jogo Live for Speed, um acessório que imite o *cockpit* de um carro de corrida e que vibre conforme as ações realizadas no jogo contribuiria para um maior nível de imersão do jogador.

Uma outra extensão seria o desenvolvimento de uma solução para a questão da injeção em múltiplos processos anteriormente relatada.

Outro uso que pode ser feito dessa ferramenta, que exige um pouco mais de conhecimento, seria o desenvolvimento de *cheats*[8]. Porém, dependendo do tipo de *cheat*, outras DLLs deverão ser interceptadas. No caso de um *WallHack*, que permite que o jogador tenha visão de raio-x e veja através de paredes, a DLL `D3D9.dll` do módulo `Direct3D9` deverá ser interceptada e os métodos responsáveis pela renderização de texturas modificados de forma a não carregar as devidas texturas, deixando a parede transparente.

[8] *Cheats* - códigos para obter privilégios especiais e modificações em valores nos jogos, com fins de trapaça.

REFERÊNCIAS BIBLIOGRÁFICAS

BRAINERD, Wade. **APIHijack** - a library for easy DLL function hooking. [S.l.], 2000. Disponível em: <http://www.codeproject.com/KB/DLL/apihijack.aspx>. Acesso em: 22 out. 2009.

CHOI, Sam-ha; CHANG, hee-Dong; KIM, Kyung-Sik. Development of force-feedback device for PC-game using vibration. In: ACM SIGCHI INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTER ENTERTAINMENT TECHNOLOGY, 74., 2004, Singapore. **Proceedings…** New York: ACM, 2004. p. 325-330. Disponível em: <http://portal.acm.org/citation.cfm?id=1067343.1067391&coll=Portal&dl=GUIDE&type=series&idx=SERIES10714&part=series&WantType=Proceedings&title=AICPS&CFID=28101901&CFTOKEN=74479414>. Acesso em: 25 mar. 2009.

FORRESTER, Justin E.; BARTON, Barton P. An empirical study of the robustness of Windows NT applications using random testing. In: USENIX WINDOWS SYSTEMS SYMPOSIUM, 4th, 2000, Seattle. **Proceedings…** Seattle: [s.n.], 2000. Não paginado. Disponível em: <http://www.usenix.org/events/usenix-win2000/technical.html>. Acesso em: 28 out. 2009.

HAGGAG, Muhammad. **Direct3D hooking sample**. Redmond, 2005. Disponível em: <http://www.gamedev.net/community/forums/topic.asp?topic_id=359794>. Acesso em: 20 out. 2009.

HUNT, Galen; BRUBACHER, Doug. Detours: binary interception of Win32 functions. In: USENIX WINDOWS NT SYMPOSIUM, 3rd, 1999, Seattle. **Proceedings...** Seattle: [s.n.], 1999. p. 1-9. Disponível em: <http://research.microsoft.com/pubs/68568/huntusenixnt99.pdf>. Acesso em: 08 out. 2009.

LANGWEG, Hanno. Building a trusted path for applications using COTS components. In: PROCEEDINGS OF NATO RTO IST PANEL SYMPOSIUM ON ADAPTIVE DEFENCE IN UNCLASSIFIED NETWORKS, 2004, Toulouse. **Proceedings...** Toulouse: [s.n.], 2004. Não paginado, paper 21. Disponível em: <http://www.rta.nato.int/Pubs/RDP.asp?RDP=RTO-MP-IST-041>. Acesso em: 28 out. 2009.

LIVE FOR SPEED. **Online racing simulator**. United Kingdom, 2009. Disponível em: <http://www.lfs.net>. Acesso em: 01 nov. 2009.

MARSHALL, Damien; WARD, Tomas; MCLOONE, Séamus. From chasing dots to reading minds: the past, present, and future of video game interaction. In: ACM CROSSROADS 13.2: COMPUTER ENTERTAINMENT, 2006, USA. **Proceedings...** USA: ACM, 2006. Não paginado. Disponível em: <http://www.acm.org/crossroads/wikifiles/13-2-CE/13-2-13-CE.html>. Acesso em: 25 maio 2009.

MATOS, Teófilo. **Introdução ao COM:** Component Object Model. Porto: [s.n.], 2004. Disponível em: <http://www.dei.isep.ipp.pt/~tmatos/ADAV/2004_2005/GuiaoCOM.pdf>. Acesso em: 28 maio 2009.

MICROSOFT. **Introdução ao DirectX**. Redmond, 2009a. Disponível em: <http://msdn.microsoft.com/pt-br/library/cc518041.aspx>. Acesso em: 26 maio 2009.

_____. **About Hooks**. Redmond, 2009b. Disponível em: <http://msdn.microsoft.com/en-us/library/ms644959%28VS.85%29.aspx>. Acesso em: 11 out. 2009.

_____. **Introduction to DirectInput**. Redmond, 2009c. Disponível em: <http://msdn.microsoft.com/en-us/library/ee418273%28VS.85%29.aspx>. Acesso em: 17 out. 2009.

_____. **Understanting DirectInput**. Redmond, 2009d. Disponível em: <http://msdn.microsoft.com/en-us/library/ee418998%28VS.85%29.aspx>. Acesso em: 18 out. 2009.

_____. **IDirectInput Interface8**. Redmond, 2009e. Disponível em: <http://msdn.microsoft.com/en-us/library/ee417799%28VS.85%29.aspx>. Acesso em: 26 out. 2009.

_____. **IDirectInputDevice8 Inferface**. Redmond, 2009f. Disponível em: <http://msdn.microsoft.com/en-us/library/ee417816%28VS.85%29.aspx>. Acesso em: 26 out. 2009.

_____. **IDirectInputEffect Interface**. Redmond, 2009g. Disponível em: <http://msdn.microsoft.com/en-us/library/ee417936%28VS.85%29.aspx>. Acesso em: 26 out. 2009.

PALMENÄS, Pontus. **Methods for stealth malware using process code injection in Windows Vista**. Sweden: [s.n.], [2008?]. Disponível em: <http://pontus.palmenas.se/articles/stealth_malware_using_process_code_injection-palmenas.pdf>. Acesso em: 23 out. 2009.

PARISI, Rodolfo. Viagem terapêutica. **Quatro rodas**, São Paulo, n. 598, p. 121, dez. 2009.

RABER, Jason; LASPE, Eric. Emulated breakpoint debugger and data mining using detours. In: PROCEEDINGS OF THE WORKING CONFERENCE ON REVERSE ENGINEERING, 14th, 2007, Washington. **Proceedings...** Washington: IEEE Computer Society, 2007. p. 271-272. Disponível em: <http://portal.acm.org/citation.cfm?id=1339502#>. Acesso em: 28 maio 2009.

SABAU, Florin. **Linha de código**: Windows Hooks. [S.l.], 2003. Disponível em: <http://www.linhadecodigo.com.br/Artigo.aspx?id=92>. Acesso em: 11 out. 2009.

MILLER, Matt; TURKULAINEN, Jarkko. **Remote library injection**. [S.l.], 2004. Disponível em: <http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf>. Acesso em: 29 out. 2009.

STAROSCIK, Matt. **Jogando com o Windows XP**: encante-se com o DirectX. Redmond, 2003. Disponível em: <http://www.microsoft.com/brasil/windowsxp/using/games/expert/staroscik_03january27.mspx>. Acesso em: 26 maio 2009.

SUN, Hung-Min; TSENG, Yu-Tung; LIN, Yue-Hsun. Detecting the code injection by hooking system calls in Windows kernel mode. In: PROCEEDINGS OF THE 2006 INTERNATIONAL COMPUTER SYMPOSIUM, 2006, Taipei. **Proceedings…** Taipei: [s.n.], 2006. p. 862-867. Disponível em: <http://dspace.lib.fcu.edu.tw/handle/2377/3598>. Acesso em: 3 nov. 2009.